From: M. Laurson. PATCHWORK: A Visual Programming Language and some Musical Applications. Studia musica no.6, doctoral dissertation, Sibelius Academy, Helsinki, 1996., chapter 5.1

• CHAPTER 5
PWCONSTRAINTS

5.1 INTRODUCTION

In this chapter we give an example of a large PW application, *PWConstraints*. It is a rule-based programming language used to solve complex musical problems. These are typically situations where the user wants to describe the end result using several analytical standpoints. Writing counterpoint is a good example. The end result is described with the help of rules controlling melodic lines, harmony, voice-leading, etc. Like all user-libraries, PWConstraints is loaded on top of PW.

The chapter is divided into four main sections. Section 5.1 offers background information and examines the main features of PWConstraints from a user's point of view.

First we see where PWConstraints sits within the field of different programming strategies. We analyse it as a constraint satisfaction problem language and refer to a number of other related environments (section 5.1.1).

When introducing PWConstraints features themselves, our starting point will be the concept of a *search-space*. We see how it is defined and how it affects the end result (section 5.1.2). We then discuss how to write PWConstraints rules. We implement a *pattern-matching scheme* that allows us to extract information needed by the rules (section 5.1.3).

In sections 5.1.4 and 5.1.5 we study several concrete examples, either general or specific in musical nature.

Section 5.2 is mostly a technical discussion. We first examine in detail the implementation of the backtrack search-engine (sections 5.2.1). The compilation of

pattern-matching rules to ordinary Lisp functions is discussed in section 5.2.2. In section 5.2.3 we introduce an important extension, allowing us to represent a *partial solution* as CLOS objects. Section 5.2.4 introduces a new type of PWConstraints rule, allowing the user to search for "better" results. We then investigate the possibility of adding the PWConstraints rule formalism to another existing CSP language (section 5.2.5). The interface between PWConstraints and PW is discussed in section 5.2.6. Finally, in section 5.2.7, we address efficiency issues by implementing a forward checking algorithm.

There then follows a discussion on two large-scale case studies (sections 5.3 and 5.4, respectively).

In the first of these, we start by studying the problem of translating counterpoint rules to the PWConstraints formalism. The rules are drawn from counterpoint text books. Section 5.3.1 gives a preparatory discussion by making comparisons between some corresponding aspects of PWConstraints and the *CHORAL* system given in Ebcioglu (1992). We then discuss the representation of musical information, extending a simple queue structure to a full-scale score representation scheme (section 5.3.2). The actual translation of the counterpoint rules, observing a number of melodic, voice-leading, harmonic, etc. considerations, is done in section 5.3.3. The extended score representation scheme requires both an extended search-space specification and a new version of the PW interface. These are given in sections 5.3.4 and 5.3.5, respectively.

The function of the latter case study is to harmonise a precomposed melodic line according to a set of rules. The basic aspects of this application, called *Harmonic Sequence Generator*, or HSG, are discussed in sections 5.4.1 and 5.42. A user-definable harmonic library, providing the basic chord materials for HSG, is then examined in section 5.4.3. The design of the actual search problem will be discussed in two parts. In the former part (sections 5.4.4 and 5.4.5), we first examine the generation of a three-part contrapuntal "skeleton" for the final result. We then go into a bit more detail, defining a number of rules controlling melodic intervals, repetitions, aspects of voice-leading, harmonic progressions, etc. The second part investigates how the contrapuntal skeleton is to be turned into a final musical result (section 5.4.6). After implementing the interface between HSG and PW (section 5.4.7) we are ready to produce an actual musical example, harmonising a melodic line by the composer Paavo Heininen (section 5.4.8).

This chapter is concluded by a short discussion that attempts to evaluate PWConstraints (section 5.5).

5.1.1 PWConstraints in Perspective

5.1.1.1 Background

When using a procedural programming language, such as C or Pascal, or a functional language, like Lisp or PW, the user has to solve a problem in a stepwise manner.[1] In many cases this approach is an adequate one, but for many types of problem it may lead to programs that are difficult to design or understand.

Descriptive (declarative) languages, like Prolog, offer an alternative way to look at this problem: instead of trying to solve a problem step-by-step, the user describes a possible result with the help of a set of rules. It is then up to the language to find solutions that are coherent with the descriptions.[2] This approach is probably more natural for individuals with a musical background. A typical music-theoretical writing offers a discussion on some properties of some pieces of music, not a step-by-step description of how those pieces were made.

PWConstraints can be thought of as a descriptive language. When using it we do not formulate stepwise algorithms, but define a search-space and produce systematically potential results from it. Typically we are not interested in all possible results, but filter (or, rather, *constrain*) these with the help of rules describing an acceptable solution.

Besides the advantages, the descriptive languages may have also some disadvantages. First, in most cases one has to specify an extremely large search-space. As a result, one has to be able to write highly efficient rules to find solutions within reasonable time. Furthermore, rules should be functional also with *partial solutions*. As it turns out, achieving this is sometimes difficult. Finally, if the rules are too strict or conflict with each other, there is no guarantee of finding a solution at all.

A specific problem in the musical domain is that the task of describing a musical result is far from trivial. As such a result is typically seen from many different points of view, we need highly flexible data structures to describe musical structures. Each rule should be able to dynamically extract required information out of a data structure, allowing the rule to analyse a result from its own point of view. One of the main problems in formulating such rules is to find a clear formalism with which to point to the required data objects.

---

[1] A general discussion of imperative, procedural, functional, declarative and object oriented languages is found in Norvig (1992:434-435).

[2] Clocksin and Mellish (1984:VII).

5.1.1.2 PWConstraints as a Constraint Satisfaction Problem Language

PWConstraints is an attempt to solve some of the descriptive language problems identified above. It is a *constraint satisfaction problem* (CSP) language, containing a number of powerful computational tools such as the *backtrack search-engine*, the *pattern-matching interface* and the *forward checking* algorithm.[3]

A CSP consists of a set of *search-variables*, each of which must be instantiated in a particular *domain* and a set of *rules* (or *predicates*) that the *values* of the search-variables must simultaneously satisfy.[4] We will call the set of search-variables, each having its domain, a *search-space*. A CSP is most often solved by using a backtrack search.

Constraint satisfaction problems already have a long history in the field of AI.[5] One of the main trends in CSP research has been the development of tools improving the efficiency of a pure backtrack search. In principle, tools like these should be as general as possible, i.e. they should not be domain specific.

In contrast to this general trend in CSP research, our discussion will be highly domain specific. Whenever we can, we will optimise a search by using the musical knowledge we have about the desired result. Although this approach is not as general as that of the classical CSP, we believe that both domain specific and non-domain specific optimisations are needed when trying to solve complex compositional tasks in an efficient way.

5.1.1.3 Related Work

Other constraint-based languages devoted to musical problems include Camilo Rueda's *Situation*, developed at IRCAM, and Kemal Ebcioglu's CHORAL project.[6]

---

[3] Although backtracking is used extensively in PWConstraints, it is not our intention here to offer a general discussion on other types of search methods (neural networks, genetic algorithms, etc.). At an early state of the PWConstraints project, genetic algorithms were in fact tested together with the backtracking approach. The results, however, were not encouraging. For a discussion on genetic algorithms, see Goldberg (1989).

[4] Mackworth (1977:99). We will use the concept "search-variable" instead of "variable" used by Mackworth, as the latter term will be reserved for the pattern-matching part of our discussion.

[5] The interested reader can follow this history in journals like Artificial Intelligence. These publish CSP-related studies regularly.

[6] *Situation* is described in Rueda and Bonnet (1993b), CHORAL in Ebcioglu (1992).

Like PWConstraints, Rueda's Situation is also written in Common Lisp and has a backtrack search-engine, a set of predefined rules and a pattern-matching language. The last-mentioned is intended to help the user in giving arguments to existing rules in a flexible way. Situation is designed specifically for generating chord sequences. It has a PW interface and allows the user to build new rules by using PW-boxes. Recently, Situation has also been used to generate metric rhythms, guitar fingerings, etc.

CHORAL will be examined in more detail in section 5.3.1, when we discuss the problem of translating counterpoint rules to the PWConstraints formalism.

Besides the above-mentioned examples, there is also a general-purpose constraint-based language implemented in Common Lisp, called *SCREAMER*. It can replace the search-engine part of PWConstraints. SCREAMER will be described in section 5.2.5, after some necessary concepts have been introduced.[7]

5.1.2 Defining a Search-Space

Solving a search problem in PWConstraints is typically done in three steps: (1) a search-space is defined; (2) a set of rules is written; (3) the search-engine is run by giving the search-space and the rules as arguments.

We will proceed with the help of some simple musical examples. In the first one of these, the task is to find all possible three-note successions in a certain search-space.

Let us start by defining the search-space. Each of the three notes (or search-variables) can be either C4, D4 or E4. In other words, C4, D4 and E4 constitute the domain of each of the three notes. The domain can also be given in MIDI-values: `60`, `62` and `64`. For practical reasons we will from now on use the latter standard.

Technically we can say that we have assigned the MIDI-value list `(60 62 64)` to each of the three notes. All solutions to our problem are found by taking all possible three-note paths. This is achieved by selecting *one* MIDI-value at a time from each MIDI-list. More formally, we can say that we take the *Cartesian product* of all domains. The *number of possible paths* is calculated by multiplying the lengths of the given domains together. In this case there are three lists, each of them containing three "candidates". Altogether we have 3*3*3 = 27 paths.

---

[7] For more information about SCREAMER, see Siskind and McAllester (1993).

```
N1   N2   N3

60   60   60
62   62   62
64   64   64
```

Fig.5.1: The search-space of a three-note succession.

Figure 5.1 gives the search-space of our example. It consists of three notes (search-variables) `N1`, `N2` and `N3`. The domains are given as a column under each note.

All possible paths (result lists) can be found for example as follows: first we pick the first candidate of each note, producing the result list `(60 60 60)`. The next list is found by taking the first candidates of `N1` and `N2` and the second of `N3`, producing the list `(60 60 62)`. Next we take the first candidates of `N1` and `N2` and the third of `N3`: `(60 60 64)`. The first candidate of `N1`, the second of `N2` and the first of `N3` produce `(60 62 60)`, etc. The entire set of paths is given in figure 5.2.

```
(60 60 60) (60 60 62) (60 60 64)
(60 62 60) (60 62 62) (60 62 64)
(60 64 60) (60 64 62) (60 64 64)
(62 60 60) (62 60 62) (62 60 64)
(62 62 60) (62 62 62) (62 62 64)
(62 64 60) (62 64 62) (62 64 64)
(64 60 60) (64 60 62) (64 60 64)
(64 62 60) (64 62 62) (64 62 64)
(64 64 60) (64 64 62) (64 64 64)
```

Fig.5.2: The Cartesian product of a three-note problem.

The search in our first example had two specific notions to it: the search-engine was run without any rules and we asked for all solutions.

Let us take a new example by extending the previous one. We now have a search-space of five notes, each note having the domain `(60 62 64 65 67 69 71 72 74 75 77)`:

```
N1   N2   N3   N4   N5

60   60   60   60   60
62   62   62   62   62
64   64   64   64   64
65   65   65   65   65
67   67   67   67   67
69   69   69   69   69
71   71   71   71   71
72   72   72   72   72
74   74   74   74   74
75   75   75   75   75
77   77   77   77   77
```

Fig.5.3: A five-note search-space.

The length of each domain being 11, the number of solutions is 11*11*11*11*11 = 161051. The first path is `(60 60 60 60 60)`, the second `(60 60 60 60 62)`, the third `(60 60 60 60 64)`, etc.

If we then extend our five-note example so that each note would have the domain `(60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77)`, the number of all paths would be 18*18*18*18*18 = 1889568. Having a 20-note example with these same 18-value domains would produce 1274823621639607817443737 6 paths.

As can be inferred from these enormous numbers, the user has to be very careful in defining the search-space.

On the other hand, a Cartesian product is very rich in possibilities: it contains all ordered pitch successions within the constraints of the search-space. Furthermore, a musical problem is often easy to translate into a search-space, and representing the search-space with the domain lists is also very economical. We need only 20 lists, each containing 18 items, to describe a search-space producing the above mentioned number of paths.

5.1.2.1 Random Ordering

Next we examine some variations on how a search-space can be defined. Until now we have looked at all solutions. Often it is enough to get only the first solution. In this case the order of the candidates affects the result. For instance we can randomly reorder each domain of figure 5.3. One possible ordering is given in figure 5.4:

```
N1   N2   N3   N4   N5

74   67   74   62   74
67   64   69   65   77
65   72   67   69   60
75   62   77   77   62
64   71   60   67   65
60   74   75   60   75
71   69   72   71   67
62   65   62   75   69
69   75   64   72   64
72   60   65   74   72
77   77   71   64   71
```

Fig.5.4: Five note search-space after reordering.

Now the first path is `(74 67 74 62 74)` and not `(60 60 60 60 60)` as in figure 5.3. This is an interesting feature because it allows us to easily get results which are

probably quite different, even though we use the same basic search-space and the same set of rules.

5.1.2.2 Preference Ordering

Ordering a search-space gives us a way to prefer some values.

N1	N2	N3	N4	N5
60	64	67	72	77
62	65	69	71	75
64	62	65	74	74
65	67	71	69	72
67	60	64	75	71
69	69	72	67	69
71	71	62	77	67
72	72	74	65	65
74	74	75	64	64
75	75	60	62	62
77	77	77	60	60

Fig.5.5: A sorted search-space.

In figure 5.5 the domain of N1 is ordered so that we always try first out values that are near to 60. The domain of N2 is sorted to be near 64, N3 to 67, N4 to 72 and N5 to 77. The overall effect of this ordering is that there is a strong tendency to produce ascending pitch successions.

5.1.2.3 Constraining Individual Notes

Another interesting variation is to constrain some notes in advance.

N1	N2	N3	N4	N5
74	67	72	62	74
67	64		65	77
65	72		69	60
75	62		77	62
64	71		67	65
60	74		60	75
71	69		71	67
62	65		75	69
69	75		72	64
72	60		74	72
77	77		64	71

Fig.5.6: A five note search-space with the third note constrained to 72.

In figure 5.6 we have constrained the third note to 72 (C5). This means that the third note will be C5 in all possible solutions. This feature allows us to "compose" part of the solution in advance and ask the search-engine to fill in the unknown parts. Now the number of possible paths is reduced to 11*11*1*11*11 = 14641.

5.1.2.4 Moulding a Search-Space

As a last example we can force the five note succession to ascend by "moulding" the domains as in figure 5.7.

```
N1   N2   N3   N4   N5

60
62   62
64   64   64
     65   65   65
     67   67   67   67
          69   69   69
          71   71   71
               72   72
               74   74
                    75
                    77
```

Fig.5.7: Five note ascending succession.

The number of paths in figure 5.7 is 3*4*5*6*7 = 2520 instead of 161051. Of course one can have many variations on this idea: descending, ascending-descending, making holes in the search-space, etc.

5.1.3 Writing Rules

Using the search-engine without any rules is of course not interesting because there are typically too many solutions and most of these are probably totally irrelevant. One of the central problems in a constraint based language is how the user can write rules in an efficient, clear and compact way without having to know in detail how the actual search-engine has been implemented. This is of course a crucial question because it is in the rules that the user defines the musical identity of the result.

PWConstraints provides a standard protocol for writing rules. Figure 5.8 gives the structure of a PWConstraints rule:

(<pattern-matching part> <Lisp-code> <doc string>)

Fig.5.8: The structure of a PWConstraints rule.

The rules are written always in three parts. We start with a *pattern-matching part*. Next we write a *Lisp-code part*, that is typically using information extracted by the pattern-matching part. The Lisp-code is a test function that either returns `true` or `nil`. As a final step we write a documentation string (*doc string*).

5.1.3.1 Partial Solutions

In order to understand how PWConstraints rules work during a search we will now take a close look on how the backtrack search-engine produces candidates for the rules. To assist the further discussion we repeat in figure 5.9 the search-space of the three note problem:

```
N1   N2   N3

60   60   60
62   62   62
64   64   64
```

Fig.5.9: Three note succession search-space.

When we explained above how the search-engine produces solutions we simplified somewhat by saying that it makes the first solution by picking the first items from each search-variable: `(60 60 60)`. What actually happens is that the search-engine does not make complete solutions in one go, but builds up the solution step-by-step. After this it calls the rules that either accept or reject the new candidate. This means that the rules are typically working with *partial solutions*.

The other difference is that instead of picking only the first accepted candidate of a given search-variable and going immediately to the next one, the search-engine checks all possible values of the search-variable in one phase and collects the accepted ones in a list. The first item of this list is stored as the current accepted value and the rest of the list is stored in a buffer. This buffer will be used later if the search runs in a dead-end situation and we have to come back (or backtrack) to the search-variable. In this case the rules are not run again because each value in the buffer is equally valid at the current state of the search. The first

item is simply popped from the buffer and the old accepted value is replaced with this item.

We call the current accepted candidate as *value* and the buffer that contains all other accepted candidates as *other-values*.

We will next simulate step-by-step the three note problem by assuming a rule that does not allow duplicates in the result:

```
             N1            N2            N3

1.step        ↓
value        60            ()            ()
other-values (62 64)       ()            ()

2.step                     ↓
value        60            62            ()
other-values (62 64)       (64)          ()

3.step                                   ↓
value        60            62            64
other-values (62 64)       (64)          ()     succeed! (60 62 64)

4.step                     ↓
value        60            64            ()
other-values (62 64)       ()            ()

5.step                                   ↓
value        60            64            62
other-values (62 64)       ()            ()    succeed! (60 64 62)

6.step       ↓
value        62            ()            ()
other-values (64)          ()            ()

7.step                     ↓
value        62            60            ()
other-values (64)          (64)          ()

8.step                                   ↓
value        62            60            64
other-values (64)          (64)          ()   succeed!  (62 60 64)

9.step                     ↓
value        62            64            ()
other-values (64)          ()            ()

10.step                                  ↓
value        62            64            60
other-values (64)          ()            ()   succeed!  (62 64 60)

11.step      ↓
value        64            ()            ()
other-values ()            ()            ()

12.step                    ↓
value        64            60            ()
other-values ()            (62)          ()

13.step                                  ↓
```

```
value         64          60          62
other-values  ()          (62)        ()    succeed!  (64 60 62)

14.step                   ↓
value         64          62          ()
other-values  ()          ()          ()

15.step                               ↓
value         64          62          60
other-values  ()          ()          ()    succeed!  (64 62 60)
```

Fig.5.10: The simulation of the three note problem.

The search starts by selecting N1 as the current note (see the small arrow pointing at N1 in figure 5.10). Next we loop through the domain of N1, (60 62 64). We check each time that the partial result has no duplicates. At N1 the partial solution contains only one item meaning that all items are accepted. The first accepted item, 60, is written as a value and the rest of the accepted items (62 64) as other-values (step 1).

Now we proceed to N2 (step 2) and loop through the domain of N2. The partial solution is built by collecting all values of previous notes as a list. Then we add to the *tail* of this list the first item from the domain of the current note. The first partial solution is (60 60) that fails because it contains duplicates. The next partial solution, (60 62), succeeds. Also the next one, (60 64), succeeds. Now there are two accepted values for N2, (62 64), and we write 62 as value of N2 and (64) as other-values.

We go to step 3 and loop through the domain of N3. We first collect the values of previous notes (60 62) and add to the tail of this list 60. The partial solution, (60 62 60), fails. Also the next one, (60 62 62), fails. The next one, (60 62 64), succeeds. This is also the first complete solution because we are pointing to the last note. If we look only for one solution we would now be finished with the search.

If we want to find more solutions we have to backtrack to the previous note N2 (see step 4). We disregard the old value and write the first item of other-values as the new value: value of N2 is now 64. It is important to note that the other-values list of N2 is now (). Also we do not have to run any rules in this case, because 64 was already accepted previously as a valid value.

We go now to step 5. We point now at N3 and collect all previous values and loop through the domain of N3. The first partial solution, (60 64 60), fails. The next one, (60 64 62), succeeds and it is also a complete solution because we are pointing to the last note. We continue and take the next partial solution, (60 64 64), that fails.

Next we backtrack again to N2 (step 6). But now we can not select a new value because other-values of N2 is (). This means that we have to backtrack to N1 and write a new value 62 for N1.

In step 7 we find two acceptable partial solutions: (62 60) and (62 64). We write as value 60 and as other-values (64).

In step 8 we find one acceptable partial solution: (62 60 64) which is also our third complete solution.

In step 9 we backtrack to N2 and write 64 as the new value of N2. We go to N3 (step 10) and succeed with (62 64 60). This is our fourth complete solution.

To get more solutions we have to backtrack to N2. N2 has no new values because other-values of N2 is (). We backtrack to N1 (step 11) and select 64 as the new value for N1.

In step 12 we proceed to N2 and find two acceptable partial solutions (64 60) and (64 62). We continue to N3 and find the fifth complete solution, (64 60 62), (step 13).

To find more solutions we backtrack to N2 and select 62 as the new value for N2 (step 14). We proceed to N3 and find the sixth complete solution, (64 62 60), (step 15).

We could still backtrack to N2 to find more solutions but we have no items in other-values of N2. The same situation occurs also when we backtrack further to N1. This means also that we have to stop the search because we can not backtrack beyond N1.

This technique of interleaving partial solutions - produced systematically by the search-engine - with the testing of those solutions by the rules, makes it possible to get results in a reasonable time even when the search-space is very large.[8] The rules should reduce the size of the search-space as early as possible. In our example the rule that disallowed duplicates was already able to make choices with partial solutions containing only two items.

Backtracking provides us with a way of undoing partial solutions that lead to dead-end situations. This is an important feature because we have to make choices before we know the complete solution.

---

[8] Of course this does not *guarantee* that we will get a result. As we will see later in this chapter the success of a search depends on many things like the search-space, the rules, etc.

5.1.3.2 Pattern-Matching

Pattern-matching has already a long tradition in artificial intelligence. Maybe the most famous example is a program called *ELIZA*.[9] The main idea is to give an illusion that ELIZA understands a conversation with the user. This can be accomplished by defining a library of pairs of patterns and responses. For instance, let us assume the following pattern-response pair:

```
pattern: (* my mother *)
response: (Tell me more about your mother ?)
```

The character "`*`" in the pattern is a *wild card* and matches to anything. The string "`my mother`" is a constant part of a sentence and the second "`*`" is also a wild card. Any sentence given by the user containing the words "`my mother`" would match to the pattern and would trigger the response "`Tell me more about your mother?`".

In the following pattern-response pair the pattern contains a *variable* (`?X`) that matches to any single word in a sentence that begins with "`I need a`":

```
pattern: (I need a ?X)
response: (What would it mean to you if you got a ?X ?)
```

For instance in the sentence "`I need a car`" the variable `?X` is bound to "`car`". This sentence would trigger the response "`What would it mean to you if you got a car?`".

The pattern-matching language of PWConstraints is a variation of the main ideas behind the pattern-matching in ELIZA. The input to the pattern-matching part of a PWConstraints rule comes from the search-engine that is systematically producing new partial solutions.

5.1.3.3 Pattern-Matching Syntax

We start by giving the complete syntax of our pattern-matching language for the pattern-matching part and for the Lisp-code part of a PWConstraints rule:

---

[9] ELIZA was developed in the 60s by Joseph Weizenbaum at MIT. In ELIZA the computer imitates a psychotherapist. The user, in turn, is the patient and types sentences in plain English. ELIZA tries to respond to this input in a reasonable way. For details see Norvig (1992:151).

```
Pattern-matching part:
?1          = variable
?           = anonymous-variable
*           = wild card
i1          = index-variable


Lisp-code part:
(?if <test>)   = begins a Lisp expression
l               = partial solution
rl              = reversed partial solution
len             = length of the partial solution
```

Fig.5.11: The pattern-matching syntax in PWConstraints.

5.1.3.3.1 Variable

A *variable* name begins always with the character "`?`" and continues with a string that can contain numbers or letters or both. Thus `?1`, `?2`, `?X`, and `?Foo1` are all valid variable names, whereas `X?1` and `?` are not, because the first one does not begin with the character "`?`" and the second one contains only the character "`?`". Variable names are unique and should be mentioned in the pattern-matching part of the rule only once. So the expression `(?1 ?2 ?3 <Lisp-code>)` is a valid rule because it contains each variable name only once, whereas `(?1 ?2 ?1 ?3 <Lisp- code>)` is not correct, because the variable `?1` is mentioned twice.

5.1.3.3.2 Anonymous-Variable

An *anonymous-variable* is given only by the character "`?`". It can appear in the pattern-matching part several times. The difference between an anonymous-variable and a variable is the following: when a rule contains variables, each variable is bound to a unique value and the name of the variable can be used in the Lisp-code part, whereas an anonymous-variable is never bound to a value i.e. it only acts as a "place-holder" in the pattern.

Let us examine two concrete examples. We assume that the search-engine has produced a partial solution consisting of the list `(1 2 3 4 5)`. To define a rule that accesses the two last values of the list, we write the following expression: `(? ? ? ?1 ?2 <Lisp-code>)`. It will bind `?1` to `4` and `?2` to `5` (figure 5.12). The three anonymous-

variables are used only to specify the place of the two last values, otherwise we are not interested about their bindings.

```
input:   (1  2  3  4  5)
pattern: (?  ?  ? ?1 ?2)

match:     ?1 = 4, ?2 = 5
```

Fig.5.12: Pattern-matching of `(1 2 3 4 5)` and `(? ? ? ?1 ?2)`.

In the second example we access the first and the fourth value of the list `(1 2 3 4 5)` with the following rule: `(?1 ? ? ?2 ? <Lisp-code>)`. This rule binds `?1` to `1` and `?2` to `4` (figure 5.13):

```
input:   ( 1  2  3  4  5)
pattern: (?1  ?  ? ?2  ?)

match:     ?1 = 1, ?2 = 4
```

Fig.5.13: Pattern-matching of `(1 2 3 4 5)` and `(?1 ? ? ?2 ?)`.

5.1.3.3.3 Wild Card

A *wild card* is always represented by the character "`*`". It can match to any continuous part of a list, this part can also be an empty list or `()`. It is important to note that a pattern can contain *only one* wild card at a time. This is in contrast with many other pattern-matching languages that allow several wild cards.

A wild card is typically used in combination with variables. For example if we have a rule, `(* ?1 <Lisp-code>)`, and match it with a list `(1 2 3 4 5)`, the wild card part of the list is `(1 2 3 4)` and `?1` is bound to `5` (figure 5.14). This means that the variable `?1` will always be bound to the last value of a given list independent of the length of the list.

```
input:   (1  2  3  4  5)
pattern: (*           ?1)

match:     * = (1  2  3  4), ?1 = 5
```

Fig.5.14: Pattern-matching of `(1 2 3 4 5)` and `(* ?1)`.

If the rule is written as `(* ?1 ?2 <Lisp-code>)`, then `?2` is bound to the last item of a list and `?1` to the second to last item. So for instance matching `(* ?1 ?2`

<Lisp-code>) with (1 2 3 4) will produce the following result: * = (1 2), ?1 = 3 and ?2 = 4 (figure 5.15):

```
input:   (1  2  3  4)
pattern: (*     ?1 ?2)

match:     * = (1  2), ?1 = 3, ?2 = 4
```

Fig.5.15: Pattern-matching of (1 2 3 4) and (* ?1 ?2).

If we try to match (* ?1 ?2 <Lisp-code>) with the list (1), we have a problem because there are not enough items in the list to bind both ?1 and ?2. In such cases PWConstraints never runs the Lisp-code part of a rule, because all variables in the pattern-matching part are not bound. This is a convenient feature because the pattern-matching part of a rule not only helps us to access useful information from a partial solution, but also decides *when* a test function will run.

5.1.3.3.4 Index-Variable

An *index-variable* name begins with the character "i". The second part of the name is a number indicating the *absolute* position of the index-variable in a given list (we start counting from one). Let us assume that we are interested in accessing the fifth, the seventh and tenth item of a list. In this case we write a rule (i5 i7 i10 <Lisp-code>). We can of course write a similar rule using anonymous-variables, (? ? ? ? ?1 ? ?2 ? ? ?3 <Lisp-code>), but this is not as compact and clear as the example using index-variables.

The pattern-matching part should not mix index-variables with variables, anonymous-variables or wild card.

5.1.3.3.5 Lisp-Code Part

Next we discuss the pattern-matching syntax of the Lisp-code part of a rule (see figure 5.11 above). Inside the Lisp-code we can use the variable names that were bound in the pattern-matching part. Also the Lisp-code part can use three special reserved variables called l, rl and len.[10] l is the partial solution (including the current candidate) found so far by the search-engine. rl is the same list but in

[10] The short names l, rl and len were chosen to make the rules as compact as possible.

reversed order. `l` and `rl` can always be accessed inside the Lisp-code part independent of the pattern matching part of the rule. `len` gives the length of the partial solution. The expression `(if? <test>)` is used to begin the Lisp-code part of the rule.

5.1.3.4 PWConstraints Pattern-Matching Compared with Other Pattern-Matchers

Our syntax may seem somewhat limited compared with other pattern-matching algorithms. For instance many implementations allow several wild cards in a pattern-matching expression. There are two main reasons for our limited or "minimal" syntax.

Firstly, we have to be very concerned about efficiency as the rules are typically called very often. This means that the pattern-matching part of a rule should be as fast as possible. Pattern-matchers that allow several wild cards are typically written as recursive Lisp functions and are too slow for our purposes. The pattern-matching part of a PWConstraints rule, however, can be compiled very efficiently.[11]

Secondly, we should have a clear and compact way of writing rules. Our main aim is to capture the most "obvious" cases in the pattern-matching part of a rule.[12] Typical pattern-matchers deal with a static stream of data (consisting for example of sentences, strings, symbols, numbers). In PWConstraints the situation is more complex - we have to match partial solutions that are constantly shrinking or expanding. Allowing multiple wild cards in such a situation, mixing index-variables with other variables and wild cards, etc., is likely to lead to a confusing system: the pattern-matching part may become extremely difficult to understand and to control.

Where we are not able to solve a special case with our limited syntax in the pattern-matching part, we can always refer in the Lisp-code part to the special variables `l` and `rl`. These allow us to write any appropriate Lisp code to extract the required information.

---

[11] See section 5.2.2.

[12] What is "obvious" in this context is of course a moot point since the main problem - describing a musical result - can be a very complex one. Our strategy is based on experience of writing dozens of rules for a wide range of musical problems.

5.1.3.5 Rule Examples

Let us first define a PWConstraints rule that disallows two adjacent equal values in a result. We write the rule in two steps: (1) the pattern-matching part; (2) the Lisp-code part. First we formulate the rule so that it better meets our requirements: the last two values in a partial solution should not be equal.

The pattern-matching part of the rule can be defined as follows: `(* ?1 ?2 <Lisp-code>)`. This means that we access always the last two adjacent items of a list irrespective of the length of the list. For instance if the partial solution is `(60 62)`, `?1` is bound to `60` and `?2` to `62`; if the partial solution is `(60)`, the Lisp-code part will not run, because `?1` is not bound; if the partial solution is `(60 62 64)`, `?1` is bound to `62` and `?2` to `64`, etc.

It is sufficient for the rule to check only the last two items of the partial solution, because this list is built step-by-step and the rule is run each time a new candidate is written at the tail of the partial solution.

Next we define the Lisp-code part. It must always start with the expression `(?if <test>)`. After running the pattern-matching part, the rule is able to access the values of both `?1` and `?2`. The Lisp-code part is a test function that returns either `true` or `nil`. If it returns `true`, the rule succeeds and the partial solution is accepted. If it returns `nil` then the rule fails and the last value of the partial solution will be rejected.

We write the Lisp-code part of the rule, which is as follows: `(?if (/= ?1 ?2))`. It simply means that `?1` and `?2` should not be equal.

The final step is to combine the pattern-matching part with the Lisp-code part to obtain the complete rule:

```
(* ?1 ?2 (?if (/= ?1 ?2)) "No adjacent notes equal")
```

Let us examine another rule that disallows duplicates in the result:

```
(* ?1 (?if (not (member ?1 (rest rl)))) "No duplicates")
```

The pattern-matching part is `(* ?1 <Lisp-code>)`, meaning that `?1` is bound to the last item of the partial solution. Thus, for instance, if this list is `(1 2 3 4)`, then `?1` is bound to `4`.

The Lisp-code part is given by the expression: `(?if (not (member ?1 (rest rl))))`. We check if `?1` is *not* a member of `(rest rl)`. If this is true, we have no duplicates of `?1` in the partial solution and the rule succeeds. `rl` gives the partial solution in reversed order. For instance, if the partial solution is `(1 2 3 4)`, then `?1`

is 4, rl is (4 3 2 1) and (rest rl) is (3 2 1). Next we check if 4 is *not* a member in the list (3 2 1), which returns true.

It is sufficient to ask if the last candidate is not a member in (rest rl), because the partial solution is built step-by-step and the rule is run for each new partial solution.

For our final example let us consider a rule which requires that neither of the first two items of a result list should be duplicated in the rest of the list. The rule is written as follows:

```
(?1 ?2 * ?3 (?if (and (/= ?1 ?3) (/= ?2 ?3)))
  "?1 and ?2 differ from the rest").
```

?1 and ?2 are the first and second items in a partial solution, as they are found *before* the wild card. ?3 is always the last item whenever the length of the partial solution exceeds two. In the Lisp-code part we check that ?3 is different from ?1 and ?2.

5.1.4 Classical Constraint Examples

Next we try out our tools with three classical constraint examples. First we define a Lisp function to run the search-engine. Let us call it *PMC* (short for "Pattern Matching Constraints"):

```
(defun PMC (search-space rules
            &key (sols-mode :once) (rnd? nil) (print-fl ()) ...)
```

PMC has two required arguments. search-space consists of a list of domains for each search-variable. rules is a list of rules. The first keyword argument is used to specify if we want all solutions (:sols-mode :all) or only one solution (:sols-mode :once). :once is the default case. The keyword :sols-mode can also be a positive integer giving the number of desired solutions. The second keyword argument, rnd?, is a flag indicating whether or not the search-space is randomly reordered (by default rnd? is nil). Also the third keyword argument, print-fl, is a flag. If it is true then the index of the current search-variable is printed indicating how far the search has proceeded.

PMC first makes an instance of a search-engine and then starts the search.[13] After the search is completed, PMC returns a list of solutions.

[13] The actual implementation of the search-engine is discussed in section 5.2.1.

5.1.4.1 A Cartesian Product Example

If we run the search-engine without any rules and ask for all solutions, `PMC` returns the Cartesian product of a given search-space:

```
? (PMC   '((a b c) (a b c) (a b c))
          ()
          :sols-mode :all)  =>

((a a a) (a a b) (a a c) (a b a) (a b b) (a b c) (a c a)
 (a c b) (a c c) (b a a) (b a b) (b a c) (b b a) (b b b)
 (b b c) (b c a) (b c b) (b c c) (c a a) (c a b) (c a c)
 (c b a) (c b b) (c b c) (c c a) (c c b) (c c c))
```

5.1.4.2 Subset Indices

Subsets have many interesting musical applications. Let us look at an example in which we want to find all 3-member subsets of the list `(0 1 2 3 4)`.[14]

One way to solve this problem is to use indices: the first possible 3-member subset is found by taking the first, the second and the third item from the superset. This subset can be represented by a list of indices `(0 1 2)` (counting from zero). The next choice is found by incrementing the last index, which produces `(0 1 3)` and `(0 1 4)`. Now the last index cannot be incremented anymore because `4` already points to the last element of the list. We therefore increment the second index to obtain `(0 2 3)` and `(0 2 4)`. We again increment the second index and get `(0 3 4)`. Next we increment the first index, which gives us `(1 2 3)` and `(1 2 4)`. We increment the second index to get `(1 3 4)`. The last possible solution, `(2 3 4)`, is found by incrementing the first index again.

To formulate this algorithm as a constraint problem let us first define the search-space: there are three indices (search-variables) called `I1`, `I2` and `I3`. Each index is in the range from `0` to `4`. Thus, the search-space is defined as `((0 1 2 3 4) (0 1 2 3 4) (0 1 2 3 4))`, where each sublist represents the domain of a given search-variable (figure 5.16):

---

[14] In the general case, the *number of subsets* can be counted by the formula *n!/(m!*(n -m)!)*, where *n* is the size of the superset and *m* is the size of the subset. In our example the number of subsets is 5!/3!*(5-3)! = 10.

```
I1   I2   I3

0    0    0
1    1    1
2    2    2
3    3    3
4    4    4
```

Fig.5.16: The search-space of the 3-member subset indices.

The Cartesian product of this search-space produces lists such as `(0 0 0)`, `(0 1 1)`, `(0 1 0)`, etc. which we are not looking for. This is because there should be no duplicates in the solution, i.e. we should never index an item in the superset more than once. Also we should avoid redundant solutions like `(0 1 2)` and `(0 2 1)` which point to the same subset.[15] All these unwanted cases can be avoided by observing that all solutions to our problem are in strict ascending order. We therefore write the following rule:

```
(* (?if (apply #'< l)) "Result in ascending order")
```

`l` found in the Lisp-code part is bound to the partial solution. The expression `(apply #'< l)` returns `true` only if the items in `l` are in ascending order.

Finally, let us run the search-engine:

```
? (PMC    '((0 1 2 3 4) (0 1 2 3 4) (0 1 2 3 4))
         '((* (?if (apply #'< l)) "Result in ascending order"))
         :sols-mode :all)  =>

((0 1 2) (0 1 3) (0 1 4) (0 2 3) (0 2 4)
 (0 3 4) (1 2 3) (1 2 4) (1 3 4) (2 3 4))
```

5.1.4.3 All Permutations

Permutations give all possible orderings of a list. For instance, let us produce all permutations of the list `(0 1 4 6)`.[16]

We find the permutations of `(0 1 4 6)` by first defining the search-space. There are four search-variables (`N1`, `N2`, `N3` and `N4`) and each search-variable has the domain `(0 1 4 6)` (figure 5.17):

---

[15] Note that we are not interested in the order of the items in a subset.

[16] In the general case, the *number of permutations* is given by *n!* where *n* is the length of the list. In our example the length is 4 so we will get 1*2*3*4 = 24 results.

```
N1   N2   N3   N4

0    0    0    0
1    1    1    1
4    4    4    4
6    6    6    6
```

Fig.5.17: The search-space of the list `(0 1 4 6)`.

The search-engine returns the Cartesian product, but to get the desired result we have to add a rule that disallows duplicates in the result. This algorithm works only for sets, i.e. the list should not contain duplicates. The "`No duplicates`" rule was explained earlier in the text, so we just restate it here:

```
(* ?1 (?if (not (member ?1 (rest rl)))) "No duplicates")

? (PMC    '((0 1 4 6) (0 1 4 6) (0 1 4 6) (0 1 4 6))
          '((* ?1 (?if (not (member ?1 (rest rl)))) "No duplicates"))
           :sols-mode :all)  =>

((0 1 4 6) (0 1 6 4) (0 4 1 6) (0 4 6 1) (0 6 1 4)
 (0 6 4 1) (1 0 4 6) (1 0 6 4) (1 4 0 6) (1 4 6 0)
 (1 6 0 4) (1 6 4 0) (4 0 1 6) (4 0 6 1) (4 1 0 6)
 (4 1 6 0) (4 6 0 1) (4 6 1 0) (6 0 1 4) (6 0 4 1)
 (6 1 0 4) (6 1 4 0) (6 4 0 1) (6 4 1 0))
```

All of our examples (Cartesian product, subset indices and all permutations) can be programmed in a normal procedural or functional way, although it is probably not easy to find algorithms that are equally elegant and simple as we found here using the search-engine. What is more important though is that both subsets and permutations are examples of problems where the amount of solutions grows extremely fast as the problem gets larger: for instance, to take all the 6-member subsets of a set of 50 items will produce 15890700 results; to make all permutations of a 12-element set will produce 479001600 results, etc. In procedural or functional programming there is no natural way to filter the search-space when the results are generated by the algorithm. Contrastingly, in constraint languages the user can reduce a large search-space simply by adding new rules. These rules are typically able to make choices already from partial solutions given by the search-engine.

5.1.5 Musical Examples

5.1.5.1 Constraining Chords[17]

Let us assume the task of calculating all possible 12-note chords where each chord is constrained by two rules: (1) the possible simultaneous intervals between adjacent notes (or *adjacent note intervals)* are given by a list of allowed intervals; (2) no pitch-class duplicates are allowed.[18]

We start by defining the search-space for 12 notes. Each note has a domain consisting of the whole piano register or in MIDI-values in the range from `21` to `108` (figure 5.18 below shows the beginning of this idea). But this approach would produce a truly large search-space.

```
N1   N2   N3   N4   N5   N6   N7   N8   N9   N10   N11   N12

21   21   21   21   21   21   21   21   21   21    21    21
22   22   22   22   22   22   22   22   22   22    22    22
..   ..   ..   ..   ..   ..   ..   ..   ..   ..    ..    ..
..   ..   ..   ..   ..   ..   ..   ..   ..   ..    ..    ..
```

Fig.5.18: The search-space of a 12-note chord.

A more sensible solution is to realise that the chord will have "gaps" or "holes" in the search-space depending on the given adjacent note intervals. For instance, let us assume that the MIDI-value of the first note is `24` and the allowed adjacent note intervals are `5` and `6`.[19] This means that the next MIDI-value of the chord has to be between `24+5 = 29` and `24+6 = 30`, i.e. the only values that we have to consider for the second note are `29` and `30`. We continue the same reasoning for the third note. We know that the *lowest* MIDI-value of the second note is `29` and we add to this number the *smallest* interval, `5`, that gives us the low limit of the third note `29+5 = 34`, etc. In this manner we can reduce drastically the search-space.

Figure 5.19 shows the reduced search-space where the allowed adjacent note intervals are `(5 6)`.

---

[17] Simultaneous note collections are henceforth called "chords".

[18] This example and the examples in sections 5.1.5.4 and 5.1.5.6 were suggested to the author by Paavo Heininen.

[19] Intervals are given as semitones.

```
N1   N2   N3   N4   N5   N6   N7   N8   N9   N10   N11   N12

24   29   34   39   44   49   54   59   64   69    74    79
     30   35   40   45   50   55   60   65   70    75    80
          36   41   46   51   56   61   66   71    76    81
               42   47   52   57   62   67   72    77    82
                    48   53   58   63   68   73    78    83
                         54   59   64   69   74    79    84
                              60   65   70   75    80    85
                                   66   71   76    81    86
                                        72   77    82    87
                                             78    83    88
                                                   84    89
                                                         90
```

Fig.5.19: The reduced search-space of a 12-note chord for the adjacent note intervals `(5 6)`.

We define a Lisp function, *make-note-ranges*, to obtain the reduction:

```
(defun make-note-ranges (start ints card) ...)
```

`start` is a MIDI-value for the first note, `ints` is the list of allowed adjacent note intervals and `card` gives the number of notes of the chord. `make-note-ranges` returns a list of lists, each sublist giving the possible MIDI-values for each note.

Next we define two rules. The first one disallows pitch-class duplicates in the chord:

```
(* ?1 (?if (not (member (mod ?1 12) (rest rl)
                        :key #'(lambda (n) (mod n 12)))))
        "No pitch class duplicates")
```

Each time we have a new value we check that its pitch-class, `(mod ?1 12)`, is not a member of the list of pitch-classes of the previous values.[20]

The second rule performs the following check: each time we get a new pair of values, the interval between them, `(- ?2 ?1)`, should be a member of the allowed adjacent note intervals `(5 6)`:

```
(* ?1 ?2 (?if (member (- ?2 ?1) '(5 6)))) "Interval rule")
```

Finally, we run `PMC` (the search-space is calculated by the `make-note-ranges` function) and ask for all possible 12-note chords with adjacent note intervals `(5 6)`:

[20] Note the partial solution consists of MIDI-values.

```
? (PMC   (make-note-ranges 24 '(5 6) 12)
       '((* ?1 (?if (not (member (mod ?1 12) (rest rl)
                         :key #'(lambda (n) (mod n 12)))))
            "No pitch class duplicates")
         (* ?1 ?2 (?if (member (- ?2 ?1) '(5 6))) "Interval rule"))
       :sols-mode :all)   =>

((24 30 35 41 46 52 57 63 68 74 79 85)
 (24 29 34 40 45 51 56 62 67 73 78 83)
 (24 29 34 39 44 50 55 61 66 71 76 81)
 (24 29 34 39 44 49 54 59 64 69 74 79))
```

5.1.5.2 A PC-Set-Theoretical Example[21]

The notion of *chains* is found in Morris (1983:432) and (1987:90).[22] Suppose that we have a list of pitch-classes. We group it into sublists where the size of each sublist is given by n. Each sublist should overlap, so that the last $n/2$ pitch-classes of a given sublist are also found at the beginning of the next sublist.[23] For example, let us assume that we have a list of 12 pitch-classes and that n is 6:

```
(n1 n2 n3 n4 n5 n6 n7 n8 n9 n10 n11 n12)
```

The list is grouped to the following sublists:

```
((n1 n2 n3 n4 n5 n6) (n4 n5 n6 n7 n8 n9) (n7 n8 n9 n10 n11 n12))
```

We will constrain our problem so that the set-class identity of each sublist should belong to a given list of set-classes. Also, if a set-class is found several times, all instances should be different member sets of that set-class.

---

[21] We will not explain any standard pitch-class set-theoretical concepts that are used in this study. Interested readers can refer for instance to Forte (1973), Straus (1990) or Castrén (1989). We use in this chapter the Tn-classification (Morris 1987:78). The set-class names are from Forte (1973), and the extensions "a" and "b", which allow us to distinguish between inversionally related set-classes, are from Castrén (1994). At times we give the prime form of a set-class after the set-class name. The prime form is written in brackets, the pitch-classes being separated by commas. For instance:
`4-1 [0,1,2,3].`

[22] We will not give details of the actual chains algorithm given in Morris (1987:92) mainly for two reasons. Firstly, we are only concerned in the chains example as a compositional idea, i.e. we want only to describe the end result, not how it is constructed. Secondly, the approach presented by Morris is a typical step-by-step algorithm. As we focus in this chapter on constraint satisfaction problems, the algorithm given by Morris is beyond the scope of our presentation.

[23] Morris uses the term "two-partition" to describe the sublists of a chain. Also he uses the term "string" to describe the complete chain. We prefer using the terms "sublist" and "list" as all our functions described below work with lists.

To implement this rule we need several new Lisp functions. First we define a function that groups a given list to sublists, called *group-to-chain*:

```
(defun group-to-chain (l card) ...)
```

`l` is the list to be grouped and `card` is the size of the sublists.[24]

```
? (group-to-chain '(1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18
                    19 20 21 22) 6)  =>

((1 2 3 4 5 6) (4 5 6 7 8 9) (7 8 9 10 11 12) (10 11 12 13 14 15)
 (13 14 15 16 17 18) (16 17 18 19 20 21) (19 20 21 22))      25
```

Next, we need a Lisp function that determines the set-class identity of a list of MIDI-values:

```
(defun SC-name (midis) ...)

? (SC-name '(0 1 2 3 4 5)) => 6-1
```

Finally, we define a function called *check-chain?:*

```
(defun check-chain? (list card SC-names) ...)
```

`list` is the partial solution given by the search-engine, `card` is the size of the sublists and `SC-names` is a list of set-classes.

`check-chain?` first groups `list` by calling `group-to-chain`. Then it calculates the last groups' set-class by calling the function `SC-name` and checks that this set-class is a member of `SC-names`. Also it checks that we do not find a given set-class several times with identical transposition. The latter check occurs only when the length of the last sublist is equal to `card`. If all checks are true then `check-chain?` returns `true` otherwise `nil`.

To be efficient we should calculate all subset-classes of the allowed set-classes in advance. Having a list of subset-classes allows us determine already from a partial solution of two or three pitches whether it potentially is able to form one of the given set-classes.[26] To accomplish this we define a Lisp function called *all-subs*:

---

[24] We are assuming that the overlapping length is `card/2`. One could make of course variations of this idea by having different overlapping lengths.

[25] As can be seen from the result the last sublist can also be shorter than `card`.

[26] Another reason why we must use subset-classes is that the last sublist returned from `group-to-chain` may be shorter than `card`.

```
(defun all-subs (set-classes) ...)
```

`all-subs` returns a list of all subset-classes of the argument `set-classes` and removes any duplicates, for instance:

```
(all-subs '(4-z15a 4-z15b))   =>

(3-3a 3-5a 3-7b 3-8b 4-z15a 1-1 0-1 2-1 2-2 2-3 2-4 2-5 2-6 3-3b 3-5b 3-7a
3-8a 4-z15b)
```

Let us next implement a chain rule where the set-classes `(6-5a 6-5b 6-18a 6-18b)` are to be found in the chain. We first store all subset-classes of the allowed set-classes under the global variable `*subs*`:

```
(defparameter *subs* (all-subs '(6-5a 6-5b 6-18a 6-18b)))
```

The rule itself is then defined as follows:

```
(* (?if (check-chain? l 6 *subs*)) "Check chain")
```

There are no variables in the pattern-matching part, because we use only the special variable `l` inside the Lisp-code part.

To make the example more interesting let us add some other rules. First, we design a rule that disallows certain set-classes within a group of three adjacent pitches. The following rule disallows the 3-member set-classes `3-9`, `3-11a` and `3-11b`:

```
(* ?1 ?2 ?3 (?if (not (eq-SC? '(3-9 3-11a 3-11b) ?1 ?2 ?3)))
    "Disallowed 3-member set-classes")
```

The pattern-matching part extracts always the three last values of a partial solution. These are passed to a function called *eq-SC?* that returns true if the set-class of `?1`, `?2` and `?3` is found in the list `(3-9 3-11a 3-11b)`. To disallow such cases we call `eq-SC?` inside `not`. `eq-SC?` is defined as follows:

```
(defun eq-SC? (set-classes &rest midis) ...)
```

In the next example we force certain set-classes to appear rather than to disallow them. We use index-variables to specify the exact positions of the given set-classes. Rules using index-variables in the pattern-match part are called *index-rules*. For instance, if we want one of the following set-classes, `(4-z15a 4-z15b 4-16a 4-`

16b), to appear at the sixth, seventh, tenth and eleventh position we write the following index-rule:

```
(i6 i7 i10 i11 (?if (eq-SC? '(4-z15a 4-z15b 4-16a 4-16b) i6 i7 i10 i11))
    "Permitted sets at indices (6 7 10 11)")
```

Let us complete the chain example by making a 24 element chain with the following rules: (1) the chain is built out of the following 6-member set-classes: (6-5a 6-18b 6-z38 6-z6 6-z43b 6-z43a 6-z41a 6-z41b 6-z12b 6-z12a 6-5b 6-z36a 6-z17b 6-z17a 6-z3a 6-18a 6-z11b);[27] (2) we disallow the following 3-member set-classes (3-9 3-11a 3-11b); (3) we force the set-class 4-1 [0,1,2,3] to appear at indices (1 2 4 8), (3 5 7 9), (6 10 11 13), (12 14 17 18), (15 16 19 20) and (21 22 23 24) - this produces in all six index-rules.

We first calculate the subset-classes of the given set-classes:

```
(defparameter  *subs*
  (all-subs '(6-5a 6-18b 6-z38 6-z6 6-z43b 6-z43a 6-z41a 6-z41b 6-z12b
              6-z12a 6-5b 6-z36a 6-z17b 6-z17a 6-z3a 6-18a 6-z11b)))
```

The search-space is defined simply by giving each element a list of all possible pitch-classes: (0 1 2 3 4 5 6 7 8 9 10 11). Finally we call PMC:

```
? (PMC
    (make-list 24 :initial-element '(0 1 2 3 4 5 6 7 8 9 10 11))
   '((* ?1 (?if (check-chain? l 6 *subs*)) "chain rule")
     (* ?1 ?2 ?3 (?if (not (eq-SC? '(3-9 3-11a 3-11b) ?1 ?2 ?3)))
         "Disallowed 3-member set-classes")
     (i1 i2 i4 i8 (?if (eq-SC? '(4-1) i1 i2 i4 i8)) "index rule1")
     (i3 i5 i7 i9 (?if (eq-SC? '(4-1) i3 i5 i7 i9)) "index rule2")
     (i6 i10 i11 i13 (?if (eq-SC? '(4-1) i6 i10 i11 i13)) "index rule3")
     (i12 i14 i17 i18 (?if (eq-SC? '(4-1) i12 i14 i17 i18)) "index rule4")
     (i15 i16 i19 i20 (?if (eq-SC? '(4-1) i15 i16 i19 i20)) "index rule5")
     (i21 i22 i23 i24 (?if (eq-SC? '(4-1) i21 i22 i23 i24)) "index rule6"))
       :sols-mode :once :rnd? t)  =>

   (8 9 7 10 4 3 6 7 5 1 0 9 2 8 3 4 10 11 6 5 2 0 11 1)
```

We ask for one solution with a randomly ordered search-space. This means that we can get a different result just by running again PMC with the same arguments.

---

27 The list was found by using a similarity measure called *RECREL* (Castrén 1994). RECREL attempts to measure the degree of similarity for two set-classes by giving a numeric value called *RECREL value*. RECREL values are in the range from 0 to 100. 0 indicates highest degree of similarity whereas 100 indicates highest degree of dissimilarity. All 6-member set-classes used in this example are within the RECREL value limits 0 and 25, inclusive, when compared to the set-class 6-5a.

Next we translate the result to MIDI-values and transpose the obtained pitch list to C4 (figure 5.20).[28] Above the staff we find a list of indices. In order to make the indexed 4-member set-classes more noticeable we transpose the notes at indices `(3 5 7 9)` and `(12 14 17 18)` one octave higher and the notes at indices `(6 10 11 13)` one octave lower. All 4-member set-classes at the given indices are analysed (the notes of each 4-member set-class are connected by braces) and we see that they all represent `4-1`.

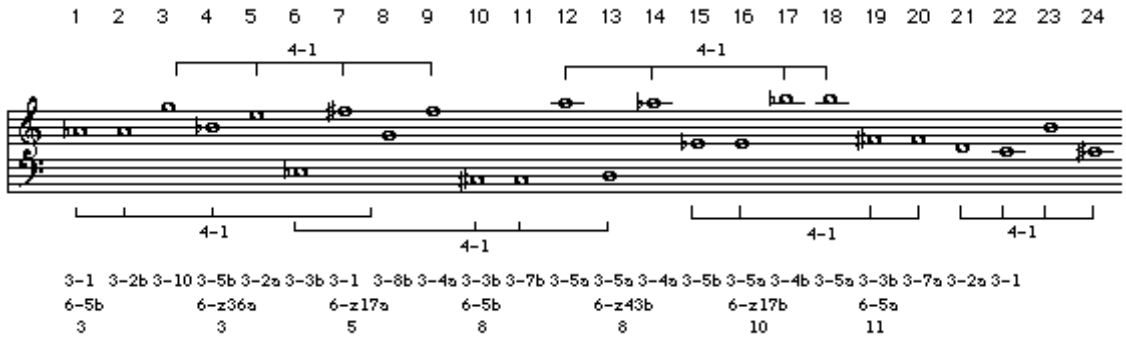

Fig.5.20: 24 element chain with overlapping 4-member set-classes `4-1`.

Below the staff, in the first line, we analyse the 3-member set-classes starting from each note. None of the disallowed set-classes `(3-9 3-11a 3-11b)` is found in this line.

In the second line the sublists of the chain is analysed. We see that all 6-member set-classes belong to the list of allowed chain set-classes. The last line gives the transposition of each 6-member set-class and we do not find a member set twice.[29]

In figures 5.21 and 5.22 we give two other chain examples. We have changed the index-rules so that in the first example we allow only set-class `4-7 [0,1,4,5]` and in the second only set-class `4-27a [0,2,5,8]`.

[28] The translation is simply done as follows: pitch-class 0 is MIDI-value 0, pitch-class 1 is MIDI-value 1, etc.

[29] Each transposition number refers to the transposition of the prime form.

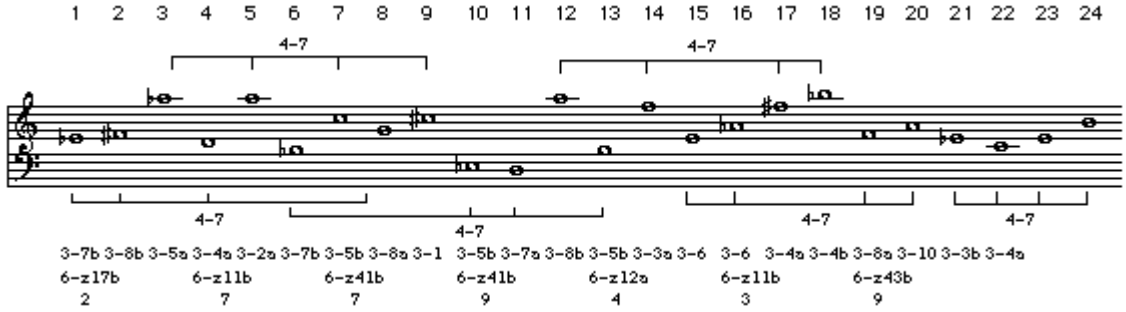

Fig.5.21: 24 element chain with overlapping 4-member set-classes `4-7`.

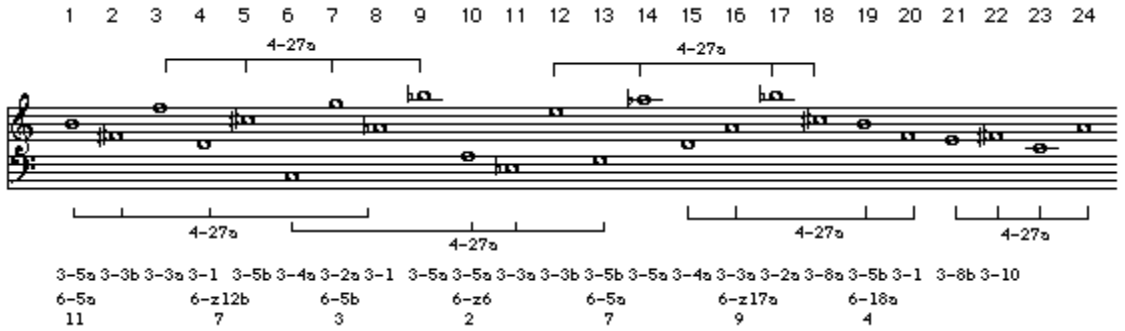

Fig.5.22: 24 element chain with overlapping 4-member set-classes `4-27a`.

5.1.5.3 Subsets

Let us suppose that we want to find all possible subsets from a list of pitches. We constrain the problem so that the set-class identities of the subsets should be members of a list of given set-classes.

We start by defining a list of pitches:

```
(defparameter  *pitches*
 '(68 69 79 70 76 51 78 67 77 49 48 81 50 80 63 64 82 83 66 65 62 60 71 61))
```

We are interested in finding all instances of set-classes `4-z15a [0,1,4,6]` and `4-z15b [0,2,5,6]` in `*pitches*`. Like in the previous example we calculate all subset-classes of `4-z15a` and `4-z15b` in advance.

```
(defparameter  *subs*  (all-subs  '(4-z15a  4-z15b)))
```

To be able to translate a list of indices to actual values we define a Lisp function called *`inds->vals`*:

```
(defun  inds->vals  (inds  ls)
  (mapcar #'(lambda (ind) (nth ind ls)) inds))
```

`inds` is a list of indices and `ls` is a list of values. For instance, if we index the sixth, first and second element of `(3 4 5 6 7 8 9)`, we get the following result (counting from `0`):

```
(inds->vals '(6 1 2) '(3 4 5 6 7 8 9))  =>  (9 4 5)
```

We start by modifying the subset indices example from section 5.1.4.2:[30]

```
? (PMC (make-list 4 :initial-element
         '(0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23))
       '((* (?if (apply #'< l)) "Result in ascending order"))
       :sols-mode :all)
```

`PMC` returns all possible indices to the 4-member subsets of `*pitches*`.

Next we add a rule that forces each subset to be either `4-z15a` or `4-z15b`:

```
(*                                                              ; 1
     (?if (let ((pitches (inds->vals rl *pitches*)))          ; 2
                 (and (not (member (mod (car pitches) 12)    ; 3
                                   (rest pitches)
                                         :key #'(lambda (n) (mod n 12))))
                      (member (SC-name pitches) *subs*))))  ; 4
    "no pc-duplicates and given SC")
```

The pattern-matching part of this rule is `(* <Lisp-code>)`, which means that this rule is run for each new value in the partial solution (1). Next we convert indices to `pitches` (2). We check that this list does not contain pitch-class duplicates (3). Then we calculate the set-class of `pitches` and check that it is found in `*subs*` (4). If we were to run `PMC` with the new rule, we would get indices to 384 subsets, each subset being a member either of `4-z15a` or `4-z15b`.

To make the example more interesting let us add two new rules to constrain the indices. For instance, we define a rule that states that the largest gap between two adjacent indices is `3`:[31]

```
(* ?1 ?2 (?if (<= (- ?2 ?1) 3)) "largest gap 3")
```

We also decide that the largest allowed gap between two adjacent indices can be found maximally only once. This is accomplished by first calculating all adjacent

---

[30] As the length of `*pitches*` is 24 each domain is defined as a list of indices ranging from 0 to 23.

[31] Note that `?1` and `?2` refer to *indices*, not to actual pitches.

index gaps. Then we check that the result contains the number 3, i.e. the largest gap, once:

```
(* ?1 ?2 (?if (<= (count 3 (mapcar #'- (cdr l) l)) 1))
  "maximally one distance of 3")
```

These two new rules will force the indices to be quite near each other. Let us put everything together and run PMC:

```
? (PMC   (make-list 4 :initial-element
          '(0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23))
          '((* ?1 (?if (apply #'< l)) "asc-order")
            (* ?1 ?2 (?if (<= (- ?2 ?1) 3)) "largest gap 3")
            (* ?1 ?2 (?if (<= (count 3 (mapcar #'- (cdr l) l)) 1))
                 "maximally one distance of 3")
            (* ?1 (?if (let ((pitches (inds->vals rl *pitches*)))
                    (and (not (member (mod (car pitches) 12) (rest pitches)
                                      :key #'(lambda (n) (mod n 12))))
                         (member (SC-name pitches) *subs*))))
                  "no pc-duplicates and given sc"))
           :sols-mode :all)   =>

((1 2 4 5) (1 3 4 6) (1 4 5 7) (4 6 9 10) (5 6 7 9) (6 8 10 12) (7 8 9 12)
 (7 9 10 11) (8 11 13 14) (9 11 14 16) (10 11 12 13) (11 12 14 17)
 (12 15 16 17) (13 15 16 19) (14 17 19 21) (15 16 17 20) (18 19 20 21))
```

The result is a list of lists of indices. We translate it to actual pitches using inds->vals, which produces the following result (figure 5.23):

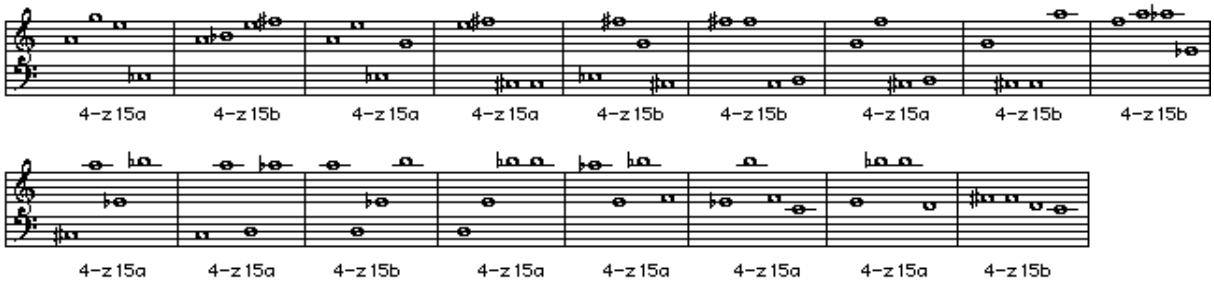

Fig.5.23: The 17 4-member subsets of *pitches*.

5.1.5.4 Statistical Distribution and Automatic Rules

Until now we have been able to define rules that simply either accept or reject a candidate. Let us, for instance, consider the following interval rule:

```
(* ?1 ?2 (?if (member (- ?2 ?1) (-1 3 -5 -4 -13 -8 -9 8 -11)))
          "Interval rule")
```

This rule first extracts the last two values of a partial solution. Then we calculate their difference and check if the interval is found in the list of allowed intervals. The problem is that this rule does not state anything about the *distribution* of the intervals. We have still no way of saying that we would like to have a lot of interval `-1`, or that interval `6` should be very rare. One solution to this problem is to make statistics of the partial solution and compare this result with the desired distribution.

The following search example is carried out in two steps. First we get the program to derive rules from an existing melodic line. These rules are inferred from the input material through statistical analysis. They are then used to make similar pitch successions by running the search-engine.

Our starting point is the soprano line from Anton Webern's Op. 16. No. 3. We extract only the pitch information in MIDI-values:

```
(68 60 71 66 61 67 72 71 66 65 76 71 63 66 77 75 74 61 70 64 78 73 60
 80 79 82 78 81 72 68 83 82 69 80 79 78 81 70 78 74 65 68 64 75 67)
```

Next we count how often each melodic interval is found in the MIDI-value list. This analysis produces the statistics given in figure 5.24 below. The result is given as a list of lists. Each sublist consists of a *count-interval pair:* we find `7` instances of interval `-1`, `5` instances of interval `3`, `5` instances of interval `-5`, etc.:

```
((7 -1) (5 3) (5 -5) (5 11) (4 -4) (3 -13) (3 -8) (2 -9)
 (1 8) (1 -11) (1 15) (1 20) (1 14) (1 -6) (1 9) (1 -2) (1 5) (1 6))
```

Fig.5.24: Interval statistics of Webern's Op. 16. No. 3.

Let us suppose that we want to create an example that has the same number of notes as the original melodic line has. In this case the rule accepts a partial solution that has *less than or equal to* `7` times interval `-1`, less than or equal to `5` times interval `3`, etc. In other words the count value determines the high limit of instances of a given interval. If, for instance, the interval `-1` is found `8` times or more, then the rule will not accept the partial solution. This rule is probably too strict and time consuming and therefore we add a *tolerance* to the rule. The tolerance indicates how much the high limit can be exceeded without failing. This means that we will get only an approximation of the desired distribution, but giving a tolerance value will speed the calculation considerably.

Next we define a Lisp function called *check-stats?*. It is general in the sense that it can be applied to any kind of data found in the original material. This data can consist of intervals, set-classes, etc.

```
(defun check-stats? (item count stats &optional (tolerance 0)) ...)
```

`item` is the data we are making statistics of. `count` indicates how many times `item` is found in the partial solution. `stats` is the statistical distribution of the original input material given as a list of count-interval pairs.

We are now ready to define a rule for interval statistics:

```
(* ?1 ?2
     (?if
      (let ((int (- ?2 ?1)))                                    ; 1
        (check-stats? int
                      (count int (mapcar #'- (cdr l) l))        ; 2
                      '((7 -1) (5 3) (5 -5) (5 11) (4 -4)       ; 3
                        (3 -13) (3 -8) (2 -9) (1 8) (1 -11)
                        (1 15) (1 20) (1 14) (1 -6) (1 9)
                        (1 -2) (1 5) (1 6))
                      1))) "Interval statistics")               ; 4
```

First we extract the two last candidates and calculate the interval between them (1). This interval, `int`, is the first argument of the function `check-stats?`. The second argument indicates how many times `int` is found in the partial solution (2). The third argument is the interval statistics found in figure 5.24 (3). The fourth argument is the tolerance which is `1` (4).

Besides the interval distribution we are also interested in the following statistics: the set-class distribution of 3-member and 4-member set-classes and statistics of four-note melodic movements in register space. The latter distribution is calculated by considering the +- *movements* of a given note group. For instance the list `(+ + -)` applied to a four note group means "up-up-down".

It is clear that making these kind of statistics by hand is quite tedious. We can automate the process by first calculating the required statistical information and then producing the Lisp text for the rules algorithmically.

The statistical analysis applied to Webern's Op. 16. No. 3 voice line produces automatically the following Lisp text:

```
(PMC
  (make-list 45 :initial-element
   '(60 61 62 63 64 65 66 67 68 69 70 71
     72 73 74 75 76 77 78 79 80 81 82 83))
  '((* ?1 ?2
       (?if
        (let ((int (- ?2 ?1)))
          (check-stats? int
                        (count int (mapcar #'- (cdr l) l))
                        '((7 -1) (5 3) (5 -5) (5 11) (4 -4)
                          (3 -13) (3 -8) (2 -9) (1 8) (1 -11)
                          (1 15) (1 20) (1 14) (1 -6) (1 9)
                          (1 -2) (1 5) (1 6))
                        1))) "Interval statistics")
    (* ?1 ?2 ?3
       (?if
        (let* ((scs (SC-distribution 3 l))
               (sc (car (last scs))))
          (check-stats? sc
                        (count sc scs)
                        '((7 3-3b) (6 3-1) (5 3-5b) (4 3-3a)
                          (3 3-10) (3 3-2a) (3 3-2b) (3 3-4b)
                          (2 3-4a)(2 3-5a) (1 3-12) (1 3-7b)
                          (1 3-8a) (1 3-11b) (1 3-9))
                        1))) "3-member set-class statistics")
    (* ?1 ?2 ?3 ?4
       (?if
         (let* ((scs (SC-distribution 4 l))
                (sc (car (last scs))))
          (check-stats? sc
                        (count sc scs)
                        '((5 4-1) (4 4-3) (3 4-12b) (3 4-6)
                          (2 4-19b) (2 4-12a) (2 4-27b) (2 4-2a)
                          (2 4-8) (2 4-5b) (2 4-9) (1 4-7)
                          (1 4-4a) (1 3-3b) (1 4-2b) (1 4-11a)
                          (1 4-16b) (1 4-z15a) (1 4-4b) (1 3-2b)
                          (1 4-z29b) (1 4-14b) (1 4-16a) (1 4-z15b))
                        2))) "4-member set-class statistics")
    (* ?1 ?2 ?3 ?4
       (?if
        (let* ((+-lists (+-distribution 3 l))
               (+-list-ref (car (last +-lists))))
          (check-stats? +-list-ref
                        (count +-list-ref +-lists :test #'equal)
                        '((13 (- + -)) (9 (- - +)) (9 (+ - -))
                          (5 (+ - +)) (2 (- - -)) (2 (+ + -))
                          (2 (- + +)))
                        2))) "+- statistics"))
    :rnd? t) 32
```

We use in the rules `"3-member set-class statistics"` and `"4-member set-class statistics"` the function *SC-distribution:*

```
(defun SC-distribution (card l) ...)
```

32 The tolerance for `"4-member set-class statistics"` and `"+- statistics"` is 2.

`SC-distribution` analyses all set-classes found in `l`, where `l` is grouped in subsequences starting from each successive item. The length of each subsequence is given by `card`. The function `+--distribution` found in the rule `"+- statistics"` is similar, except we analyse in this case the +- movements found in `l`.

Figure 5.25 below shows on the upper staff the pitches of the original voice line. The lower staff, in turn, gives a result obtained by using automatic rules.

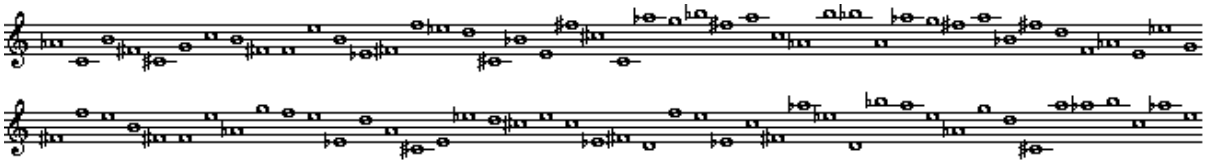

Fig.5.25: The original melodic line and a result produced by automatic rules.

To complete the example we give the interval distribution, the set-class distribution of 3-member and 4-member set-classes and +- statistics of the result (figure 5.26):[33]

```
Interval distribution:
((8 -1) (6 -5) (6 11) (4 3) (4 -13) (3 -4) (3 -8) (2 20) (1 8)
 (1 -11) (1 14) (1 -6) (1 9) (1 15) (1 -9) (1 -2))

3-member set-class distribution:
((7 3-3b) (7 3-1) (4 3-4b) (4 3-5b) (3 3-2a) (3 3-2b) (3 3-3a) (3 3-5a)
 (2 3-4a) (2 3-10) (1 3-12) (1 3-7b) (1 3-8a) (1 3-11b) (1 3-9))

4-member set-class distribution:
((6 4-3) (5 4-1) (5 4-6) (3 4-2a) (2 4-8) (2 4-16b) (2 4-27b) (2 4-4b)
 (2 4-12b) (2 4-5b) (1 4-19b) (1 3-3b) (1 4-11a) (1 4-9) (1 4-z15b)
 (1 4-16a) (1 4-z15a) (1 4-2b) (1 4-12a) (1 4-z29b) (1 4-14b))

+- movement distribution:
((13 (- + -)) (9 (- - +)) (9 (+ - -)) (5 (+ - +)) (4 (- - -))
 (1 (+ + -)) (1 (- + +)))
```

Fig.5.26: Statistics of the result.

Our example is of course somewhat artificial because we do not consider at all metric aspects, phrase boundaries, etc. of the original melody. We will come back to these problems - i.e. to problems that deal with *musical context* - in the third and fourth part of this chapter. Also we should note that in this example we used *absolute* counts. This means that our example works only if the search problem has exactly the same amount of notes than is found in the original. In the general case the

[33] The statistics of the original was given in the previous `PMC` expression.

statistical distributions should be defined as *proportional* (for instance as percentage) values. This would allow the number of notes of the search problem to differ from the original.

5.1.5.5 The "At Least" Property

The statistical rules discussed above could be called *at most* rules, because we give in the rules an upper limit of how many instances of some musical property can be found in the final solution. A related problem is how to define in the rules the *at least* property.[34]

One solution to this problem is to insist that the partial solution has the desired properties as early as possible.[35] For instance let us consider the following example. The length of a complete solution (`total-len`) is 5. We want to have at least 3 (`atleast-count`) items with a certain property in the final solution. In this case we can insist that there is at least *one* such item in a partial solution when the length of the partial solution is 3, *two* such items when the length is 4, *three* such items when the length is 5, etc.

Let us call this count as `current-atleast-cnt` and the length of the current partial solution as `current-len`. The value of `current-atleast-cnt` is calculated by the following expression: `(- current-len (- total-len atleast-count))`.

Based on this idea we define a general function, *`atleast-cnt-check`*, that checks for the at least property:

```
(defun  atleast-cnt-check
  (l total-len atleast-cnt-item-lst &optional (test #'eq)) ...)
```

`l` is the current partial solution and `total-len` is the length of the final solution. `atleast-cnt-item-lst` is a list (or a list of lists) of count-item pair(s).

For instance, let us suppose the following search example. The search-space consists of 6 search-variables, each having the domain `(0 1 2 3)`. A result should contain at least 3 zeros and at least 2 ones. Also, no adjacent duplicates are allowed in a result:

```
(PMC (make-list 6 :initial-element '(0 1 2 3))
  '((* ?1 (?if (atleast-cnt-check l (cur-slen) '((3 0) (2 1))))
```

---

[34] The at least property example was suggested to the author by Camilo Rueda.

[35] The trivial solution is of course to check for the at least property only when we have a complete solution, but this approach is very inefficient.

```
      "at least check")
    (* ?1 ?2 (?if (/= ?1 ?2)) "no adjacent dups"))
 :sols-mode :all)  =>

((3 0 1 0 1 0) (2 0 1 0 1 0) (1 0 3 0 1 0) (1 0 2 0 1 0) (1 0 1 0 3 0)
 (1 0 1 0 2 0) (1 0 1 0 1 0) (0 3 1 0 1 0) (0 3 0 1 0 1) (0 2 1 0 1 0)
 (0 2 0 1 0 1) (0 1 3 0 1 0) (0 1 2 0 1 0) (0 1 0 3 1 0) (0 1 0 3 0 1)
 (0 1 0 2 1 0) (0 1 0 2 0 1) (0 1 0 1 3 0) (0 1 0 1 2 0) (0 1 0 1 0 3)
 (0 1 0 1 0 2) (0 1 0 1 0 1))
```

We use in the `"at least check"` rule the function *cur-slen* that returns the number of search-variables in the current search-engine.

5.1.5.6 Splitter

The final example, called *splitter*, is a search problem where we distribute some musical information among two (or more) parts. For example, our starting point can be the following list of pitch-classes:

```
(defparameter *pcs* '(0 1 2 3 6 7 0 1 8 2 3 9 0 7 8 1 2 5 0 6 11 1
                      4 7 2 3 8 1 7 10 0 5 6 7 8 11))
```

We assume that this list should be split into two parts according to some rules given by the user.

Our first problem is how to construct a search-space for the splitter. One simple solution is to take each item from `*pcs*` and say that this item has to belong *either* to the first *or* to the second part. This can be accomplished by defining each value of a domain as a list of two items. This list will be called *partnum-data pair*. The first item in this pair gives the part number, while the second one contains some musical data.

For instance in our particular example the first domain of the search-space is defined as `((1 0) (2 0))`. This means that the first item of `*pcs*`, `0`, belongs *either* to part `1` *or* to part `2`. The second domain, in turn, is `((1 1) (2 1))`. This means that the second item of `*pcs*`, `1`, belongs either to part `1` or to part `2`. The third domain is `((1 2) (2 2))`, etc. We generate the search-space by the following expression:[36]

```
? (mapcar #'(lambda (n) (list (list 1 n) (list 2 n))) *pcs*)  =>
(((1 0) (2 0)) ((1 1) (2 1)) ((1 2) (2 2)) ((1 3) (2 3)) ((1 6) (2 6))
.............................................................
 ((1 8) (2 8)) ((1 11) (2 11)))
```

[36] We show due to space limitations only the beginning and the end of the search-space.

Next, we define four help functions. The first two are simple ones allowing us to extract the part number and the data of a partnum-data pair:

```
(defun partnum (partnum-data) (first partnum-data))
(defun data (partnum-data) (second partnum-data))
```

As the next help function we define the function *setp:*

```
(defun setp (list &key (test #'eq) (key #'identity)) ...)
```

`setp` returns `true` if `list` is a set, i.e. does not contain duplicates.

The fourth help function, *`data-group-of-part`*, is defined as follows:

```
(defun data-group-of-part (partnum-data-lists group-len partnum) ...)
```

`partnum-data-lists` is a list of partnum-data pairs, `group-len` indicates the length of subgroups and `partnum` is the part number of the current part.

First, `data-group-of-part` collects all partnum-data pairs that belong to the current part to a list (the current part is given by `partnum`). Then this list is grouped into sublists. The length of the sublists is given by `group-len`. Finally, `data-group-of-part` returns the data items of the first sublist.

After this we discuss an example of a splitter rule that forces certain set-classes to appear within adjacent items inside a part. We define the possible set-classes and their subset-classes as a global variable:

```
(defparameter *subs* (all-subs '(4-16b 4-16a 4-14a 4-8 4-6 4-5b)))
```

The splitter rule is defined as follows:

```
(* ?1 (?if
        (let ((pcs (data-group-of-part rl 4 (partnum ?1)))     ; 1
              (if pcs
                 (and (setp pcs) (member (SC-name pcs) *subs*)) ; 2
                  t))) "splitter rule")
```

First, we collect the most recent sublist of data items, `pcs`, using the function `data-group-of-part` (1).[37] The group length of this particular rule is `4`, i.e. we are interested only in 4-member set-classes. Finally, we check that `pcs` does not contain duplicates and that its set-class belongs to `*subs*` (2).

[37] We collect the most *recent* sublist as we use `rl`.

After this we define another rule that guarantees us that the result will be balanced in the sense that both parts are forced to alternate after a certain number of pitch-classes. For this rule we need the following help function:

```
(defun count-adjacent-items (list &optional (test #'identity)) ...)
```

count-adjacent-items counts the number of adjacent equal items found at the beginning of `list`. The equality can be defined by the user by giving an optional `test` function.

The rule itself is defined as follows:

```
(* ?1 (?if (let ((max-cnt 3))                          ; 1
            (if (>= (length rl) (1+ max-cnt))
              (<=  (count-adjacent-items rl     ; 2
                     #'(lambda (a b)
                         (= (partnum a) (partnum b))))
                   max-cnt)
              t))) "max 3 adjacent notes in the same part"))
```

`max-cnt` (here `max-cnt` is `3`) indicates the maximum number of adjacent items in a part (1). The function `count-adjacent-items` counts the number of adjacent equal items found in `rl` (2). The equality is defined as a test function that checks if two adjacent items belong to the same part.

Finally we run the splitter:

```
(PMC
  (mapcar #'(lambda (n) (list (list 1 n) (list 2 n))) *pcs*)
  '((* ?1 (?if
         (let ((pcs (data-group-of-part rl 4 (partnum ?1))))
              (if pcs
                 (and (setp pcs) (member (SC-name pcs) *subs*))
                  t))) "splitter rule")
     (* ?1 (?if (let ((max-cnt 3))
                  (if (>= (length rl) (1+ max-cnt))
                    (<=  (count-adjacent-items rl
                           #'(lambda (a b)
                               (= (partnum a) (partnum b))))
                         max-cnt)
                    t))) "max 3 adjacent notes in the same part"))
   :rnd? t) =>

((2 0) (1 1) (2 2) (2 3) (1 6) (2 7) (1 0) (2 1) (1 8) (1 2) (1 3) (2 9)
 (1 0) (1 7) (2 8) (1 1) (2 2) (1 5) (2 0) (1 6) (1 11) (2 1) (1 4) (2 7)
 (1 2) (1 3) (2 8) (2 1) (2 7) (1 10) (1 0) (1 5) (2 6) (1 7) (1 8) (1 11))
```

The result is given as partnum-data pairs. As in section 5.1.5.2 we translate the pitch-classes of the result to pitches. Figure 5.27 shows the result where the

pitches belonging to part 1 are transposed to C3 and the pitches belonging part 2 to C5:

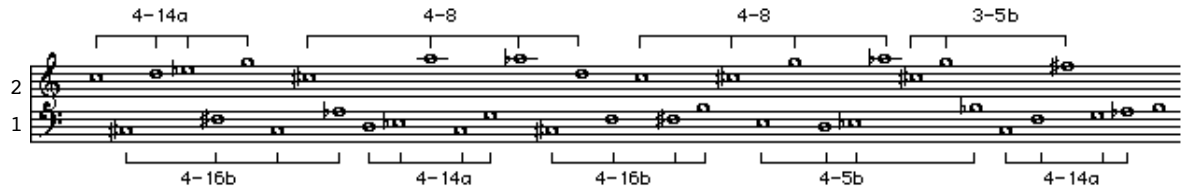

Fig.5.27: A two part split.

Our splitter-rule is general in the sense that it can split into an arbitrary number of parts. For instance we can split *pcs* in three parts by defining the search-space with the following expression:

```
(mapcar #'(lambda (n) (list (list 1 n) (list 2 n) (list 3 n))) *pcs*)
```

Also we add an extra rule that guarantees that all three parts are always present inside five successive pitch-classes:

```
(* ?1
  (?if
    (let ((size 5))
       (if (>= (length rl) size)
         (and (>= (count 1 rl :key #'partnum :end size) 1)
              (>= (count 2 rl :key #'partnum :end size) 1)
              (>= (count 3 rl :key #'partnum :end size) 1))
          t))) "at least one instance of each part inside 5 pcs")
```

Figure 5.28 shows a result of the three part split where, after translating the pitch-classes to pitches, part 1 is transposed to C3, part 2 to C4 and part 3 to C5:

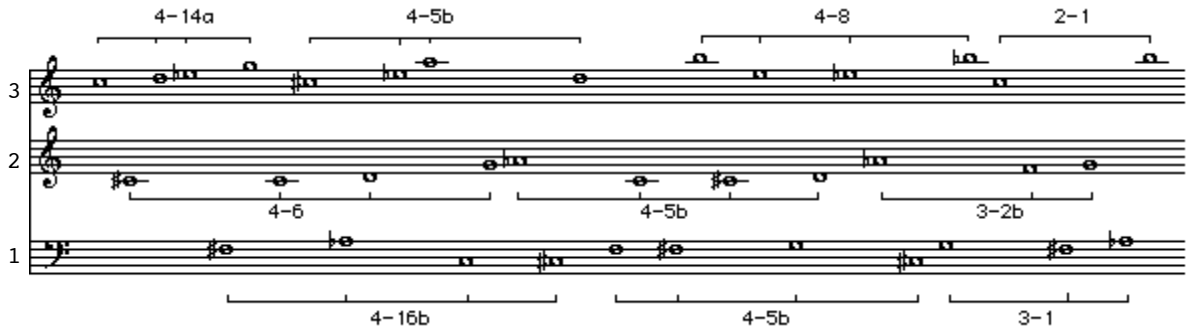

Fig.5.28: A three part split.