SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2011; **00**:1–99 *Prepared using* speauth.cls *[Version: 2002/09/23 v2.2]*

Using XQuery for problem solving

SP&E

Pekka Kilpeläinen[†]

University of Eastern Finland, School of Computing, Kuopio, Finland

KEY WORDS: XQuery; programming; problem solving; puzzles; combinatorial search

SUMMARY

XQuery is a recommendation by the World-Wide Web Consortium as a standard XML query language. In addition to being a special-purpose query language, XQuery has also features which support unexpected applications like problem solving. We demonstrate and discuss these features by presenting several XQuery programs for solving recreational problems and puzzles.

1. Introduction

Computer applications have intriguing potential for enhancing human problem-solving capabilities. They allow us to play with ideas in their nascent stage and to experiment with possible solutions. Spreadsheet calculators and scripting languages are often used as ad-hoc tools to replace tedious pen-and-pencil work in such circumstances. The author of this article has previously used script languages like AWK [1] and Python [51], or the logic programming language Prolog [13, 49] for experimenting with combinatorial ideas, and for rapid prototyping of designs, say, of virtual machine architectures, before implementing them in a procedural language like C, Ada, or Java. Recently, XQuery has in several occasions been chosen as a tool for such experiments by the author.

We demonstrate the potential of XQuery for experimental problem solving by considering XQuery solutions to several recreational problems and puzzles. We share Donald Knuth's attitude towards puzzles [33, pp. 7–9]. That is, the puzzles are not the most important thing, but they are ideal for introducing representative problems—and of course, they are fun. The

*Correspondence to: University of Eastern Finland, School of Computing, P.O. Box 1627, FI-70211 Kuopio, Finland
[†]E-mail: Pekka.T.Kilpelainen@uef.fi

purpose of presenting computerized solutions to recreational puzzles is not to spoil their charm through replacing entertainment of rational challenges by application of computing power. Simple puzzles often have generalizations which go beyond human ability. Developing instructive *methods* which could be applicable for attacking generic problems with the help of a computer is our main motivation. Problem solving methods being considered here are XQuery programming techniques, which we believe to be novel for many programmers.

As a functional query language designed for XML manipulation, XQuery may be a strange tool for those programmers who are familiar with procedural general-purpose programming languages. Indeed, XQuery has some peculiarities which cause problems for novice programmers (including the author!). For example, XQuery has a rather flexible syntax. This is because XQuery is built around XPath expressions, which are used for addressing components of XML documents. XML documents—by the very design of the Extensible Markup Language [9]—may use arbitrary tag-names, including XQuery keywords, which thus appear as subexpressions of XPath location path expressions. For such reasons, a simple syntax error often causes an XQuery compiler to give error messages whose actual cause may be difficult to locate. Therefore some of XQuery's syntactic and semantic pitfalls will be pointed out along the discussion.

The central ideas and essential features of XQuery are reviewed in Section 2. The main body of the article consists of XQuery solutions to a number of problems and puzzles. Section 3 deals with arithmetical problems whose solutions are integers which satisfy the constraints of the problem. Specific problems considered include the generation of small prime numbers, "cryptarithmetic puzzles", and a problem of selecting optimal face values for stamps. Section 4 considers examples of combinatorial arrangement puzzles, including the non-attacking n queens problem and popular puzzles known by the names "Instant Insanity", "Tricky Triangles", and "Sudoku". The applicability of XQuery as a problem-solving language is discussed in Section 5. Section 6 reviews related work, and Section 7 is a conclusion.

The XQuery programs presented in this paper have been tested with the Saxon-HE processor (v. 9.3.0.2J)[†]. Saxon is an in-memory implementation of XQuery (and XSLT) written in Java. It appears to be a faithful and robust implementation of the W3C specifications. Some measurements of Saxon executions are presented. Results of such measurements are of course specific to the implementation and the execution environment, but they can nevertheless be indicative of realistic usability of XQuery, and of the effects of some presented solution variants on their efficiency. The experiments were carried out on a single-user Intel Core 2 Duo E7500 workstation (clock speed 2.93 GHz, FSB speed 1066 MHz, 3 MB L23 cache) with a 4096 MB and 1333 MHz DDR3 Dual Channel RAM. Saxon was executed under the 64 bit Centos Linux 5 operating system using OpenJDK version 1.6.0_17. The time and memory usage of Java executions was measured using a Java agent application, which was written using the `java.lang.management` package as a small modification of an example given in [21, pp. 296–298]. The execution time was measured using `System.nanoTime()`, and memory usage as the total of the peak usage of different memory pools.

[†] `http://www.saxonica.com`

SHL

2. Essentials of XQuery

This section reviews the essentials of XQuery, restricting to central features which are used in our treatment. The W3C recommendation [7] is recommended as a comprehensive XQuery reference. (A potential reader of the Recommendation is warned that the typing rules of XQuery are rather complicated. On the other hand, its operational semantics, which we are relying on, is relatively simple.) As a readable XQuery tutorial, see, e.g., [52]. In addition to the basics discussed in this section, some additional XQuery features, like specific built-in functions are introduced in later sections while applying them to specific problems. Readers who are familiar with XQuery may likely skip to Section 2.7 without a risk of missing much information.

XQuery is a language designed for querying and manipulating XML documents, and other data sources which can be viewed as XML structures. The closest predecessor and the strongest influence for XQuery was an academic XML query language called Quilt [12]. XQuery uses XPath *location path expressions* for selecting parts of XML documents. XQuery is a proper extension of XPath 2.0 [3], which is also a part of the XSLT 2.0 transformation language [30]. In addition to the selection capabilities of XPath, XQuery also supports traditional database operations like joins and grouping, and XML transformations similar to XSLT.

The design of XQuery was guided by its anticipated applications gathered from the database and document communities. These XML query use cases collected in a W3C note [11] provide useful examples of the anticipated use of XQuery. In comparison, the present article provides examples of somewhat unanticipated applications of XQuery.

2.1. Data model

XQuery is a functional expression language. This means that its execution is based on side-effect-free evaluation of expressions. XQuery expressions may be combined basically without any limitation; of course some combinations, like trying to apply arithmetics to non-numeric operands, cause typing errors. Such an orthogonal design is grounded by a simple and streamlined type system: All XQuery expressions operate on sequences, and evaluate to sequences. *Sequences* are ordered lists of items. *Items* can be either *nodes*, which represent components of XML documents, or *atomic values*, which are instances of XML Schema base types [5] like `xs:integer` or `xs:string`. Sequences can also be empty, or consist of a single item only. No distinction is made between a single item and a singleton sequence.

XQuery and XPath 2.0 share a common data model [4], and a common set of over 100 functions and operators [38]. XQuery/XPath sequences differ from lists in languages like Lisp and Prolog by excluding nested sequences. Designers of XQuery may have considered nested sequences unnecessary for the manipulation of document contents. Nesting, or hierarchy of document structures is instead represented by nodes and their child-parent relationships; these are discussed in more detail below.

For example, the below expression evaluates to a flat sequence of six items, which are an integer 1, a string “cat”, integers 1, 2, and 3, and a decimal number 3.14.

```
((), 1, ("cat", 1 to 3), 3.14)
```

XQuery uses the comma ',' as the sequence concatenation operator. The parentheses '(' and ')' are used for grouping of expressions. In practise they are useful as delimiters of sequence expressions, because the comma operator has a low precedence, which often confuses the scope of the concatenation without the use of surrounding parentheses. An empty pair of parentheses '()' is a simple way to express an empty sequence. The expression "`1 to 3`" above is a *range expression*, which is useful for creating sequences of consecutive integers.

Components of XML documents are represented by *nodes*. There are six types of nodes, out of which the most important ones are *document*, *element*, *attribute* and *text* nodes. Nodes comprise a hierarchical structure that corresponds to the nesting of the components of an XML document. A *document node* is the root of a complete XML document tree, which contains all the contents of the document represented by its descendant nodes. *Element nodes* represent elements of an XML document. They can have other element nodes or *text nodes* as their children, representing the immediate contents of the element. XML elements may also carry *attributes*, which are uniquely named fields with a string value. The attributes of an element are represented by *attribute nodes* related to the corresponding element node.

As an example, consider the small XML document shown below:

```
<D><E a="1"><C />foo</E><E a="2" b="3">bar</E></D>
```

The corresponding XQuery/XPath tree consists of 10 nodes, which are shown as a diagram in Fig. 1. Element nodes are drawn as circles, and text nodes as boxes. Child-parent relationships are drawn as solid lines, while attribute nodes are connected with their containing element node using dashed lines. The nodes are numbered in the diagram by their position in the *document order*, which is roughly the same as the pre-order traversal of the tree. Attribute nodes come in the document order immediately after their corresponding element node. The relative order of attribute nodes of an element is immaterial and thus left implementation-dependent. For example, instead of the order shown in Fig. 1, the attribute `a="2"` could also appear at position 9 in the document order, with attribute `b="3"` at position 8. A complete XML document tree is always rooted by a document node, which has the element node of the document root element as its child (node 2 in Fig. 1). In XPath, each element and even the entire document has a text value, which is the catenation of its content strings. The value of our example document is `"foobar"`, which is the same as the value of its root element `D`.

2.2. Location path expressions

Nodes of document trees are selected using XPath *location path expressions*. A location path is evaluated with respect to a *context node*, which is often the root of a document. For this, let us assume that `example.xml` is the URI of the above XML document. Then the XPath function `doc("example.xml")` evaluates to the node 1 of Fig. 1. A location path consists of *steps* separated with slashes '/', which are evaluated to further nodes which are accessible from the context node. For example, the expression

```
doc("example.xml")/D/E
```

evaluates to the sequence that consists of nodes 3 and 7, in this (document) order. Location path expressions are evaluated from left to right, each step with respect to the nodes produced

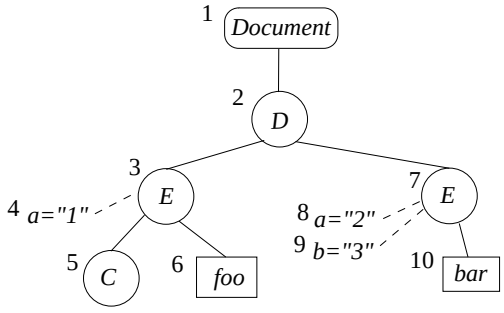

Figure 1. Example of an XPath tree

by the previous steps. (Note that this is a sketch of the *conceptual semantics* of location path expressions. Implementations may apply alternative evaluation strategies which produce equivalent results.) Each of the previously produced nodes is used, one at a time, as the context node for the next step. Node sequences produced by location paths are in the document order and do not contain contain duplicate nodes.

XPath specifies 12 different tree-relationships called *axes*, which can be used for expressing location path steps. The `child` axis does not have to be explicitly mentioned, and it is thus effectively the default axis of location path steps. Other axes allow, for example, the parent node, or the preceding or following sibling nodes of the context node to be accessed; we'll return to them in the examples when they are needed. In addition to an axis, a step always contains a *node test*, which is often simply the name of the node, like `D` and `E` above. A wild-card '`*`' matches any name. A neutral test "`node()`" accepts any node (including unnamed ones like text nodes). Text nodes can be selected by using "`text()`" as a node test. It is often useful to be able to select sub-ordinate nodes without knowing their exact level. For this the shorthand notation "`//`" which corresponds to zero or more child steps is very useful. The *attribute axis* allows the attribute nodes of an element to be accessed. The shorthand notation for an attribute step is '`@`' followed either by an attribute name or by the wild-card '`*`'.

Let us consider some examples of the above location-path constructs on the above document. The expression

```
doc("example.xml")//E
```

selects all `E` elements, which is in this case equivalent with the previous expression. The more general expression

```
doc("example.xml")//*
```

selects all elements of the document, and evaluates to the sequence of nodes (2, 3, 5, 7). All child nodes of `E` elements can be obtained with

```
doc("example.xml")//E/node()
```

which evaluates to the sequence of nodes (5, 6, 10). If we want to get only *element* children of the `E` elements, we can use

```
doc("example.xml")//E/*
```

The above expression evaluates to node 5 only. The evaluation of the last child step '`*`' with respect to node 7 produced an empty result. The flexibility of location paths causes sometimes problems: an erroneously written location path often simply evaluates to an empty result, which is not considered as an error, and is thus not reported in any special way by the implementation. (Built-in functions `exactly-one()`, `one-or-more()`, and `zero-or-one()` support the manual inclusion of explicit checks on the cardinality of expression values in XQuery programs.)

All attributes of `E` elements can be obtained by applying the node-name wild-card to their attribute axis as follows:

```
doc("example.xml")//E/@*
```

The above expression evaluates to the sequence of nodes (4, 8, 9). Similarly, attributes with name `b` within any element of the document can be obtained as follows:

```
doc("example.xml")//*/@b
```

This query selects the single attribute node 9 in Fig. 1.

2.2.1. Filter expressions

Results of location path steps can be filtered using any number of *predicates*. Predicates are Boolean conditions written in brackets '`[`' and '`]`' immediately after the step. The conditions use the candidate node, which is being tested for inclusion in the result, as their context node. Predicates can be used to filter sequences of other items than nodes, too. The item to be tested for the inclusion in the result can be expressed in both cases as a period '`.`'. For example, the integers between 1 and 100 which are divisible by 13 can be produced with a filter predicate, using the XPath remainder operator `mod`, as follows:

```
(1 to 100)[. mod 13 = 0]
```

The above expression evaluates to the sequence (13, 26, 39, 52, 65, 78, 91). Further examples of similar filter expressions are given in Sec. 3.1 below.

Predicates may use functions, which are evaluated with respect to the candidate item and the sequence in which it belongs. Such functions, like `position()`, are often useful, for example to select items by their position in the sequence. A short-hand notation similar to ordinary array indexing has been included in XPath: if the value i of a filter predicate is numeric, then the condition is treated as "`position() =` i", effectively selecting the ith item of the sequence. Sequence-valued expressions can be used as predicate conditions, too, with the interpretation that an empty sequence corresponds to the Boolean value `false()` while non-empty sequences correspond to `true()`. (XQuery does not contain names for the Boolean constants, but they can be expressed with built-in functions `true()` and `false()`.) This is often useful for selecting nodes of a document, say, based on their sub-structures.

Let us consider some examples of the above mentioned features. The first element child of any node in the document tree can be obtained with the below query:

```
doc("example.xml")//*[1]
```

The above expressions evaluates to the sequence of nodes (2, 3, 5). Correspondingly, the last child—of any type—below any node in the document can be obtained using the `last()` access function as follows:

```
doc("example.xml")//node()[last()]
```

Since `last()` is of a numeric type, it is treated here as the condition "`position() = last()`". The above expression evaluates to the sequence of nodes (2, 6, 7, 10).

If we instead want to get the last node in the entire document, we need to use parentheses to expand the scope of the predicate from the previous step to the entire expression, as follows:

```
(doc("example.xml")//node())[last()]
```

The latter query selects the node with number 10 only.

A step can contain multiple predicates, for example to locate the first of the `E` elements whose content equals "bar", as follows:

```
doc("example.xml")//E[. = "bar"][1]
```

The above evaluates to the node 7 of Fig. 1. Since the predicates are evaluated one at a time, the above is *not* equivalent with `doc("example.xml")//E[1][. = "bar"]`. The latter expression produces an empty result, since the value of the first `E` element is "foo" instead of "bar". We can locate those elements which have a child element `E` with value "foo" as follows:

```
doc("example.xml")//*[E = "foo"]
```

The above query evaluates to the document element `D`. Consider how the predicate `[E = "foo"]` is evaluated with respect to this element. The location path `E` produces a *sequence* consisting of the nodes 3 and 7. This is a typical case, which arises for example when locating article elements by the condition that the name of some of their authors has a given value. Comparison with sequences works because of the special semantics of XPath comparison operators, which are discussed below.

2.3. Comparison operators

The comparison between a sequence and a string value works in XPath because of the semantics of *general comparisons*. Ordinary comparison operators `=`, `!=`, `<`, `<=`, `>`, and `>=` are used to express *general comparisons*, which have an *existential semantics*. This means that the comparison succeeds if and only if a pair of items, one from each operand sequence, can be found such that they satisfy the requested relationship between their values. For example, the below expression evaluates to `true()`:

```
3 < (1, 4) and (1, 4) < (2, 3)
```

The first comparison is satisfied by the pair of items (3, 4), and the second one by the pair (1, 2) or (1, 3). The existential semantics of general comparisons is often useful, but it has also unexpected implications. For one thing, it destroys the transitivity of comparisons. For example, despite that the above comparisons are successful, the comparison `3 < (2, 3)` fails, since its second operand has no item that would be greater than 3.

General comparisons allow *set operations* on sequences be expressed easily. XPath includes explicit set operations `union`, `intersect`, and `except`, but they apply to *node* sequences only. Let us consider expressing set operations on sequences `$A` and `$B` of *atomic* items. (XQuery variable names are prefixed with a dollar sign. We follow this convention in examples which refer to arbitrary operands.) The *membership* of an atomic item `$a` in sequence `$A` can be expressed using general comparison, by testing whether any item of `$A` equals `$a`:

```
$a = $A
```

The above assumes that the types of `$a` and of the items in `$A` are compatible. For example, if `$a` is an integer but `$A` contains strings, this comparison may raise a typing error. This can be circumvented using a built-in function called `index-of()`. The `index-of()` function takes a sequence and an item, and returns the sequence of those positions (if any), where the item occurs within the sequence. Incompatible types are treated simply as mismatches. So, testing whether item `$a` appears in sequence `$A` can be expressed with

```
not(empty(index-of($A, $a)))
```

The meaning of the built-in functions `empty()` and `not()` should be obvious: the first one tests whether its argument is empty, and the second one returns the Boolean complement of its operand. Similarly, the *difference* of sequences `$A` and `$B` could be expressed with

```
$A[not(. = $B)]
```

(that is, leave those items of `$A` which are not equal with any item of `$B`), or more type-safely using the `index-of()` function as follows:

```
$A[empty(index-of($B, .))]
```

The *intersection* of `$A` and `$B` can be expressed similarly, either with

```
$A[. = $B]
```

or in a manner which applies also to sequences which contain values of incompatible types:

```
$A[not(empty(index-of($B, .)))]
```

Finally, the *union* of two sequences `$A` and `$B` of atomic items can be conveniently expressed using a built-in function `distinct-values()`, which eliminates duplicates from its argument, as follows:

```
distinct-values(($A, $B))
```

If there is no need to eliminate duplicates, then simple sequence catenation with `($A, $B)` is sufficient for uniting the sequences.

In addition to the general comparisons, XPath also provides corresponding *value comparisons*, which are expressed using operator keywords `eq`, `ne`, `lt`, `le`, `gt`, and `ge`. Value comparisons apply to atomic values only: they raise a type error if they are applied to sequences of more than one item. Value comparisons also require their operands to be of compatible types. So, the use of value comparisons instead of general comparisons can increase type-safetyness of programs: They allow us to catch situations, where we would be dealing with values of different cardinality or of different type than what we expect. On the other hand, general comparisons are sometimes convenient. Element and attribute values are of type `xs:untypedAtomic` unless they have been assigned a more specific type by validating them against a schema. General comparison of such untyped values against a typed operand casts the untyped value into the known type, and this makes general comparisons convenient when dealing with values drawn from non-validated XML structures.

XQuery and XPath 2.0 include typical *Boolean expressions* with operators `and` and `or`. In addition, *quantified expressions* are also included, which allow to test whether some (or every) item of a given sequence satisfies a given condition. An interesting application of quantified expressions is shown in Section 3.3.

The relationship between general comparisons and value comparison can be explained using quantified expressions. For example, the general comparison `$A != $B` is equivalent with the below quantified expression:

```
some $a in $A satisfies
     some $b in $B satisfies $a ne $b
```

That is, can we select some item `$a` from sequence `$A` and some item `$b` from sequence `$B` so that their values are different?

2.4. Element constructors

XQuery supports the creation and modification of XML structures by *element constructors*, which are somewhat similar to XSLT templates. A *direct element constructor* consists of an XML element, whose markup and contents are transformed to the corresponding nodes of an XPath tree (see Fig. 1). (There are also *computed constructors* which allow the element name to be computed by an expression.) Element constructors may contain subexpressions surrounded with braces '`{`' and '`}`'. These subexpressions are evaluated before inserting their value in the document subtree created by the element constructor. For example, assume that `$x` is a variable bound by some outer expression (like FLWOR; see below). Then the expression

```
<R attr="{$x}">Value of $x = {$x}</R>
```

creates an element `R` where the value of `$x` appears both as the value of the attribute `attr` and in the contents of the element, after the phrase "`Value of $x = `".

2.5. FLWOR expressions

Element constructors are often used in so called *FLWOR expressions*. The FLWOR expression (pronounced "flower") is the XQuery counterpart of the SQL SELECT-FROM-WHERE construct. FLWOR is the most central XQuery expression for expressing document queries. A FLWOR expression allows to select, re-arrange and create document contents (modeled by node sequences), but it can also create sequences of other kinds of items.

The letters of the acronym FLWOR stand for the parts of the expression, called *clauses*, which are `for`, `let`, `where`, `order by`, and `return`. A FLWOR expression begins by one or more `for` or `let` clauses, which introduce variable bindings to be used in the rest of the expression. Their difference is that a `for` clause introduces variables which iterate over the items of a given sequence, while a `let` clause binds its variables to an entire sequence. As an example, consider the below expression:

```
for $i in (1 to 3),
    $j in ($i + 1 to 3)
let $s := ($i to $j)
return  $s
```

The above query introduces three variables. Variable `$i` iterates over the integers 1, 2, and 3, and for each of its values, variable `$j` iterates over the integers $\texttt{\$i}+1, \ldots, 3$. Variable `$s` is bound to the integer sequence (`$i`, ..., `$j`) and used as the return value of the expression. So the result of the above consist of the sequences (1, 2), (1, 2, 3), and (2, 3). When catenated together, the result of the above expression becomes the sequence (1, 2, 1, 2, 3, 2, 3) of seven items.

A FLWOR expression creates its result sequence by evaluating its `return` clause once for each combination of variable bindings created by the `for` and `let` clauses. The variable bindings may be pruned by an optional `where` clause, and ordered by an optional `order by` clause. Consider the above expression extended by a `where` clause and an `order by` clause as follows:

```
for $i in (1 to 3),
    $j in ($i + 1 to 3)
let $s := ($i to $j)
where sum($s) gt 4
order by count($s)
return  $s
```

The built-in function `sum()` returns the sum of the atomic values of its input sequence, which must be numbers. The above `where` clause restricts the bindings to those for which the sum of the numbers in sequence `$s` exceeds four. The `order by` clause orders the binding combinations in increasing order with respect to the length of sequence `$s`. The result consists of sequences (2, 3) and (1, 2, 3), which catenated together form the final result (2, 3, 1, 2, 3).

SHL

2.6. User-defined functions

XQuery is a Turing-complete programming language. This results from allowing users to define functions, which may be recursive. Recursion is the only control structure in XQuery which supports unbounded repetition. The `for` clause provides a means of *bounded* repetition over explicitly created sequences.

User-defined functions belong to a namespace, which must be explicitly prefixed either with a user-declared namespace prefix or with the default prefix "`local:`". Consider a recursive XQuery function for computing the factorial $n! = 1 \cdot 2 \cdots (n-1) \cdot n$ of a given integer n. A function namespace can be declared as follows:

```
declare namespace f = "http://www.uef.fi/cs/XQueryTesting/Fact";
```

The function declaration can then be given as follows:

```
declare function f:fact($n as xs:integer) as xs:integer
 { if ($n le 1) then 1   (: by convention, 0! = 1 :)
   else $n * f:fact($n - 1) };
```

This declaration says that the function `f:fact()` takes an integer parameter, and returns an integer value. The body of this function consists of a *conditional expression* (`if-then-else`), which is used to specify its value in the base case and in the recursive case. One syntactic peculiarity of XQuery (and XPath) is that the `else` branch of a conditional expression is mandatory. Phrases enclosed in the "smiley faces" "`(:`" and "`:)`" are XQuery comments. User-defined functions can be used as sub-expressions of queries, for example as follows:

```
for $n in 0 to 10
return concat($n, "! = ", f:fact($n), ";")
```

The above expression uses the built-in function `concat()` to concatenate together the value of `$n`, the constant string "`! = `", the factorial of `$n`, and a semicolon, for each `$n` from 0 to 10. The result of the above expression is a sequence of 11 strings from "`0! = 1;`" to "`10! = 3628800;`".

2.7. Generating permutations

We conclude this section by a nontrivial example, which demonstrates some peculiarities of XQuery, and how they can be circumvented. Solutions of several combinatorial problems involve generating permutations, that is, different orderings, and selecting out of them those that satisfy the constraints of the problem.

We first present two fallacious solutions to the task of generating permutations, which hopefully illuminate relevant features of XQuery. The high-level principle of the solution is the following: Let S be the sequence to be permuted. If it is empty or if it contains only a single item, then S comprises its own permutations. Otherwise, the permutations of S consist of sequences that result by (1) removing, one at a time, each item s of S, (2) generating the permutations of the remaining items recursively, and (3) placing the removed element s in front of each of the computed permutations. Below is our first—erroneous—implementation of this idea as a recursive XQuery function `perm0()`:

```
declare function local:perm0($items as item()*) as item()* {
  if (count($items) le 1) then ($items)
  else for $i in 1 to count($items),
           $perm in local:perm0(remove($items, $i))
       return insert-before($perm, 1, $items[$i]) };
```

The above declaration specifies that the function both takes and returns a sequence of zero or more items. Built-in functions `remove()` and `insert-before()` are used for removing an item from a sequence by its position, and for inserting an item at a specified position. The intent is that the variable `$perm` iterates over the permutations generated from the input sequence with its `$i`th item removed, and that `insert-before($perm, 1, $items[$i])` inserts the `$i`th item of the input sequence as the first item in a permuted sequence `$perm`. Unfortunately this attempt does not work. When applied to the sequence `("a", "b", "c")` we would expect to get six permuted sequences as the result, but the function produces instead a sequence `("a", "b", "a", "c", ..., "c", "a")` of 24 strings. What goes wrong is the lack of nested sequences. Especially, the above variable `$perm` does not iterate over permuted sequences, as intended, but over items instead.

Nesting can be represented with element nodes. XPath trees made of element nodes are indeed designed for representing hierarchical data. So our next try is to wrap permutations inside `perm` elements, using the below function `perm1()`:

```
1  declare function local:perm1($items as item()*) as element(perm)+ {
2    if (count($items) le 1) then <perm>{$items}</perm>
3    else for $i in 1 to count($items),
4             $perm in local:perm1(remove($items, $i))
5         return <perm>{insert-before($perm/node(), 1, $items[$i])}</perm> };
```

Notice that the function signature has been changed to indicate that the `perm1()` function returns a non-empty sequence of `perm` elements, instead of arbitrary sequences of items. New wrapper elements are generated for the permutations on lines 2 and 5. When the variable `$perm` is bound to such an element, line 5 creates a new `perm` element, whose children consist of the item `$items[$i]` followed by the children of the `$perm` element. For example, the invocation

```
local:perm1(("a", "b", "c"))
```

produces the below sequence of six elements:

```
(<perm>abc</perm>, <perm>acb</perm>, <perm>bac</perm>,
 <perm>bca</perm>, <perm>cab</perm>, <perm>cba</perm>)
```

There is still one problem, which is not quite obvious in the above result. Even though the items are permuted correctly, their identity is lost. The contents of each of the above elements consists of a single text node only. Had the original items been strings of different length, there would be no way to extract them from the contents of the `perm` elements. XPath models contiguous fragments of textual content by a single text node. For that reason XQuery merges adjacent text nodes into a single text node by concatenating their contents, even if the nodes are inserted one by one, as above.

The last correction to our permutation generation is to preserve the identity of the items by enclosing also the items in elements, as shown below:

```
1   declare function local:permutations($items as item()*) as element(perm)+ {
2    let $itemElems := ( for $i in $items return <item>{$i}</item> )
3    return local:perm1($itemElems) };
```

The above acts as wrapper function, which simply encloses each item `$i` of its input sequence in an `item` element (on line 2), before invoking the actual permutation with the `perm1()` function declared above.

3. Arithmetic problems

In this section we present XQuery programs for solving arithmetic problems on integers. Some problems have simple solutions based on XQuery's capabilities to introduce and to filter integer sequences; these are considered in Section 3.1. The generation of prime numbers is a relevant application of these methods. A straight-forward XQuery approximation of the "sieve of Eratosthenes" is presented as a solution to the task. Section 3.2 considers "cryptarithmetic puzzles", which are equations made of words whose letters stand for individual digits. Section 3.3 considers solving a stamp-value selection problem, where the requirement is to be able to pay each requested postage using at most three stamps.

3.1. Filtering integer sequences

Some arithmetic problems have easy XQuery solutions based on filtering integer sequences. As a simple example, consider first the below Problem 1 from Project Euler[‡]:

> Add all the natural numbers below one thousand that are multiples of 3 or 5.

Combining a range expressions, a filter expression, and the `sum()` aggregate function yields the below "one-liner" as an XQuery solution to this problem:

```
sum( (1 to 999)[. mod 3 eq 0 or . mod 5 eq 0] )
```

Generation of prime numbers is a more relevant task which is also easily solved by filtering integer sequences. We consider generating all primes which do not exceed the value of an external parameter `$N`, which can be introduced as follows:

```
declare variable $N external;
```

Our first solution is a direct realization of the *definition of primes*, that is, they are integers bigger than one which are not divisible by any other prime except them self:

```
let $candidates := 2 to $N
for $cand in $candidates
where count($candidates[$cand mod . eq 0]) eq 1
return $cand
```

[‡] `http://projecteuler.net`

The above solution uses a FLWOR expression for introducing the variable `$cand` to examine each candidate number in turn. Using this variable we can apply the filter expression `$candidates[$cand mod . eq 0]` to find the divisors of `$cand` in the `$candidates` sequence, and consequently include only those candidates that do not have other divisors.

The above expression has an obvious inefficiency: a number can be a multiple of smaller numbers only, and thus it is unnecessary to test any items that come *after* `$cand` in the `$candidates` sequence as potential divisors of `$cand`. The `for` clause of a FLWOR expression allows one to declare also *positional variables*, which provide a means for realizing this optimization. A positional variable iterates over the *positions* of the sequence, synchronized with the variable which iterates over the items of the sequence. Below we declare a positional variable `$pos` (on line 2) to accept only those candidate numbers `$cand` which are not *preceded* by any of their divisors in the `$candidates` sequence:

```
1   let $candidates := 2 to $N
2   for $cand at $pos in $candidates
3   where empty($candidates[position() lt $pos][$cand mod . eq 0])
4   return $cand
```

The optimized version is clearly more efficient with respect to both time and space. The run time and the amount of main memory used by the above programs, identified as `primes.xq` and `primes-opt.xq`, are displayed for different values of `$N` = 10 000, 12 000, ..., 30 000 in Figure 2 and Figure 3. Each test was repeated five times. The results of each set of repetitions are plotted as an error bar between the minimum and maximum value, and consecutive medians are connected with lines. (The unit MB stands for 10^6 bytes in this article.)

A resource lower-bound estimate for an XQuery evaluation can be obtained from a program which consists of a single string expression like `"Hello world!"`. A Saxon evaluation of this program requires about 540 ms of time and about 28 MB of RAM in our test environment.

Prime numbers can be generated more efficiently with methods based on the well-known "sieve of Eratosthenes". Eratosthenes' method is simple: To generate all primes up to n, first initialize a list of integers $2, 3, \ldots, n$. Composite numbers are then crossed out from the list until only primes remain. More precisely, repeat the following steps

1. Declare the next uncrossed number p to be a prime, and
2. Cross out the multiples of p from the remaining list

until all numbers have either been crossed out or declared to be prime. It suffices to cross out multiples of p starting from p^2, because smaller multiples have already been crossed out at earlier iterations. Observe that each number is crossed out at most once for each of its distinct prime factors. The number of distinct prime factors of an integer n is known to be $O(\log \log n)$ [28, § 22.11]. This yields that the complexity of Eratosthenes' algorithm is $O(n \log \log n)$, which is almost linear. This complexity requires that each number can be crossed out from the list in constant time. XQuery does not support arrays or destructive assignments, which makes the algorithm difficult to realize. The below function is thus rather an approximation of the sieve of Eratosthenes known as "trial division" [43].

```
1   declare function pr:sieve($cands as xs:integer*) as xs:integer* {
2     (: Pre: $cands are in increasing order and
3              contain no multiples of any prime smaller than $cands[1] :)
```

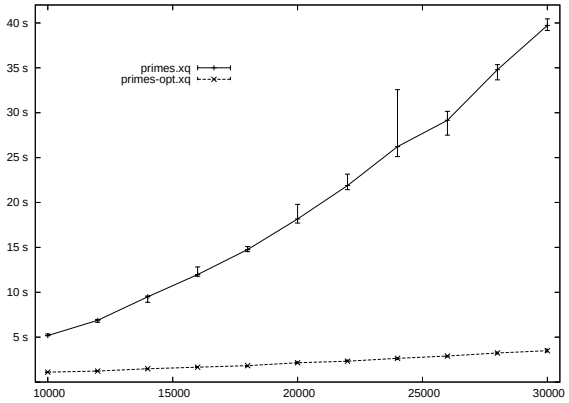

Figure 2. Execution times and ...

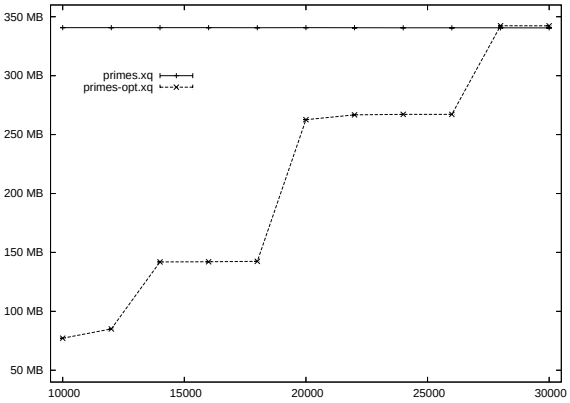

Figure 3. memory usage of `primes.xq` and `primes-opt.xq`

```
4     if ($cands[1] * $cands[1] gt $cands[last()]) then
5       $cands    (: all of $cands are primes :)
6     else ( $cands[1], pr:sieve($cands[. mod $cands[1] ne 0]) ) };
```

The function is invoked with the call `pr:sieve(2 to $N)` to compute all primes up to `$N`. The key idea of this tail-recursive function is to pass the sequence `$cands` of remaining candidate numbers forward (on line 6) without multiples of the prime which is found as its first item `$cands[1]`. Because the function scans the contents of the `$cands` parameter once for each prime number at most $\sqrt{n}$, instead of once for each candidate number, it is an order of magnitude faster than the above FLWOR expressions. For example, the function `pr:sieve()` computes the 9,592 prime numbers which are smaller than 100,000 in about 1.2 s, while it takes about 23 s for the optimized FLWOR expression to compute them. The execution time and the amount of RAM needed by function `pr:sieve()` are displayed in Figure 4 and in Figure 5, respectively, for different values of `$N`$= 100\,000, 200\,000, \ldots, 2\,000\,000$.

O'Neill has shown that the time complexity of "trial division" is $\Theta(n^{1.5}/(\log n)^2)$ for generating the prime numbers up to n [43]; a similar analysis applies also to the above `sieve()` function. She has also discussed the efficient implementation of the genuine sieve of Eratosthenes in a lazy functional language (Haskell). Generating of primes is a fascinating subject, but its further elaboration is beyond the scope of this article.

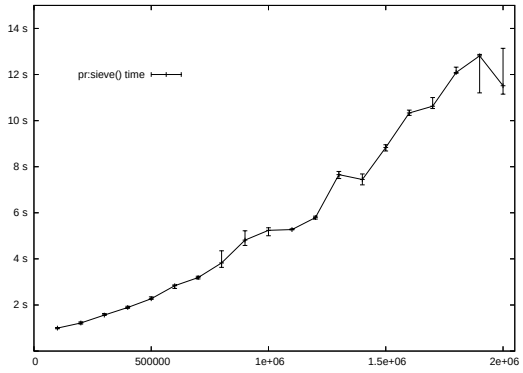

Figure 4. Execution time and ...

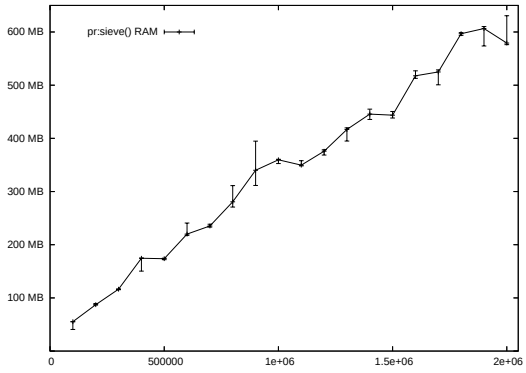

Figure 5. memory usage of the `pr:sieve()` function

3.2. Cryptarithmetic puzzles

Cryptarithmetic, or verbal arithmetic puzzles are equations made of words. The task is to find distinct digits which can be substituted for the letters so that the equation becomes true. The classic example of a cryptarithmetic puzzle, based on the addition of two 4-digit numbers, was originally published by Henry Dudeney in 1924 [33, p. 324]:

$$\begin{array}{r} \text{S E N D} \\ +\ \text{M O R E} \\ \hline = \text{M O N E Y} \end{array}$$

Let us first consider a similar but simpler problem by Polya [45, p. 234] as a preliminary for cryptarithmetic puzzles. The task of this problem is to deduce the digits represented by two blanks. Polya's problem can be stated as follows:

> An old bill indicates that 72 items were bought for the total price of $_67.9_. The first and the last digit of the sum have faded and are thus illegible. How much was the total, and what was the price of a single item? Assume that each item cost the same (integral) number of cents.

The problem can easily be solved as an application of a FLWOR expression, for example by iterating over possible initial and final digits `$a` and `$b` of the sum and testing which of them give a total which is divisible by 72:

```
for $a in 1 to 9, $b in 0 to 9
```

```
let $sum := 10000*$a + 6790 + $b
where $sum mod 72 eq 0
return ("total: ", $sum, "unit: ", $sum div 72)
```

An alternative solution is based on iterating over total prices between 16,790 and 96,799 cents, and restricting to those (1) which have the known "679" as their three middle digits and (2) which are divisible by 72:

```
for $sum in (16790 to 96799)[
  6790 <= (. mod 10000) and (. mod 10000) <= 6799  (: condition 1 :)
  and . mod 72 eq 0 ]                              (: condition 2 :)
return ("total: ", $sum, "unit: ", $sum div 72)
```

Sometimes, like above, the selection condition can be expressed within a filter expression, instead of using a `where` clause.

Let us next consider an XQuery solution to the SEND+MORE=MONEY puzzle. Blind search among possible values of letters may lead to a rather inefficient solution. For example, a straight-forward translation of a constraint logic programming based solution given in [55] lead to an XQuery program, whose evaluation took about 17 s in our test environment. A more optimized program is presented in Fig. 6. It takes about 1.3 s to produce the solution

```
 9567
+1085
------
10652
```

XQuery uses XML character entities for including characters by their Unicode position in strings. Line 1 in Fig. 6 declares a global variable `$LF`, which is used as a constant string that consists of the linefeed character only, to format the result on lines 17–20. The rest of the program consists of a single FLWOR expression, which introduces eight variables `$S`, `$E`, `$N`, `$D`, `$M`, `$O`, `$R`, and `$Y` for the distinct digits that we are looking for. The carries which may arise in the column-wise additions are modeled by the variables `$c1`, `$c2` and `$c3` (lines 5–6). The leading digit `$M` of the bottom line is the carry for the addition of the left-most column. The most essential part are the conditions on lines 11–16, which express the requirement for disjoint digits and the equations that arise by the column-wise additions.

The following slightly more complex puzzles are from [46, Sec. 2.4]:

```
    C R O S S          D O N A L D
+   R O A D S        + G E R A L D
-------------  and   -------------
= D A N G E R      = R O B E R T
```

Solving the first of them (with 9 letters) in a manner similar with the above puzzle took about the same time, while DONALD+GERALD=ROBERT (with 10 letters) took clearly longer, about 3 s. The complexity of cryptarithmetic puzzles generally increases exponentially with respect to the number of their letters. This seems inevitable: David Eppstein has shown that testing whether a cryptarithmetic puzzle has a solution is an NP-complete problem [20]. The NP-completeness requires that the puzzles use arbitrarily many letters and arbitrarily

```
1   declare variable $LF := "&#10;";  (: linefeed :)
2
3   let $M := 1, (: disallowing leading zeros, $M cannot be else :)
4       $otherDigits := (0 to 9)[. ne $M], (: digits except $M :)
5       $carries := (0, 1)
6   for $c1 in $carries, $c2 in $carries, $c3 in $carries,
7       $S in $otherDigits[. gt 0], (: again, disallow leading zeros :)
8       $E in $otherDigits, $N in $otherDigits, $D in $otherDigits,
9       $O in $otherDigits, $R in $otherDigits, $Y in $otherDigits
10  where (: require distinct digits: :)
11      count(distinct-values( ($S, $E, $N, $D, $M, $O, $R, $Y) ) ) eq 8
12      and (: satisfy the column-wise additions: :)
13            $D + $E =  $Y + 10*$c1 and
14      $c1 + $N + $R =  $E + 10*$c2 and
15      $c2 + $E + $O =  $N + 10*$c3 and
16      $c3 + $S + $M =  $O + 10*$M
17  return ( $LF, concat(" ", $S, $E, $N, $D, $LF),
18                concat("+", $M, $O, $R, $E, $LF),
19                concat("------",  $LF),
20                concat($M, $O, $N, $E, $Y, $LF) )
```

Figure 6. An XQuery solution to the SEND+MORE=MONEY puzzle

large digits. Ordinary puzzles with decimal numbers allow at most 10! different assignments of digits to letters, and thus lead to instances which are solvable in linear time, even though with a large constant factor.

3.3. A postage stamp problem

Consider the following puzzle [26, Problem 10]: Find a series of seven denominations for postage stamps such that each postage 1, 2, ..., 70 can be paid using at most three stamps. (Say, the envelopes have room for three stamps only.) This is a difficult search problem. Its solution consists of the unique series of stamp values $(1, 4, 5, 15, 18, 27, 34)$, which can be found in about 14 min by running the program shown in Fig. 7.

The solution is based on a FLWOR expression with a quantified expression in its `where` clause. The code is hand-optimized by trying to restrict the huge number of different variable bindings by a number of observations. The candidate denominations are assigned to the variables `$stamp1`, ... `$stamp7` in strictly increasing order. First, the smallest denomination `$stamp1` (line 2) must be 1 to be able to express the smallest postage. The next bigger denomination `$stamp2` (line 3) must not be greater than 4, since otherwise the postage 4 cannot be formed using at most three stamps. The upper limits of the ranges for the values of the remaining variables `$stamp3`, ..., `$stamp7` (lines 4–8) are figured out as follows: The maximum value that is needed for the highest denomination `$stamp7` is `$maxSum`$-2 = 68$, since the maximum postage 70 can be formed using three stamps as $68 + 1 + 1$. Next, the

```
1   let $maxSum := 70
2   let $stamp1 := 1
3   for $stamp2 in (2, 3, 4),
4       $stamp3 in ($stamp2 + 1 to $maxSum - 6),
5       $stamp4 in ($stamp3 + 1 to $maxSum - 5),
6       $stamp5 in ($stamp4 + 1 to $maxSum - 4),
7       $stamp6 in ($stamp5 + 1 to $maxSum - 3),
8       $stamp7 in ($stamp6 + 1 to $maxSum - 2)
9   let $options := (0, $stamp1, $stamp2, $stamp3,
10                   $stamp4, $stamp5, $stamp6, $stamp7)
11  where every $sum in (7 to $maxSum) satisfies
12          some $a in $options[. le $sum],
13               $b in $options[(. ge $a) and (. le $sum - $a)],
14               $c in $options[. eq $sum - $a - $b ]
15                  satisfies true()
16  return $options[. gt 0]  (: = $stamp1, $stamp2, ..., $stamp7 :)
```

Figure 7. An XQuery solution to the postage stamp problem

value of the second largest denomination `$stamp6` does not need to exceed `$maxSum`$-3 = 67$: if it equals 67, then the highest denomination is 68, and each postage 67,. . . , 70 can be formed using at most three stamps. A similar reasoning is applied to the upper-bounds of the ranges for the variables `$stamp5`, `$stamp4`, and `$stamp3` on lines 4–6.

Line 9 introduces the sequence `$options` of the values that can be chosen for the stamps. Value zero models the possibility of using fewer than three stamps to pay a postage. The condition that each postage must be payable using at most three stamps `$a`, `$b` and `$c` is expressed on lines 11–15. The range of required sums (line 11) starts from 7. When the second smallest denomination `$stamp2` is between 2 and 4 (by line 3), we know without testing that each postage up to 6 can be formed using at most three stamps with values `$stamp1` or `$stamp2`. Lines 12–14 generate values for three stamps in a non-decreasing order, such that their value totals to the value of `$sum`. Finally, the combinations of stamp denominations—in this case a single one only—which satisfy all the constraints are returned on line 16.

The above program examines almost 29×10^6 different combinations of denominations. Some further optimizations to reduce this number could be found. For example, it seems reasonable to start the search for the values of the largest denomination (`$stamp7`) only at 24, since the maximum postage 70 cannot be paid with three stamps whose values are smaller. Anyhow, this optimization reduced the number of considered value combinations by less than 0.2%, and did not have any clear positive effect. A heuristic which was tried was to test the target sums of the range `($7 to $maxSum)` in a reverse and thus descending order, with the intuition that a larger postage is harder to form and would therefore allow unsuccessful bindings to be rejected earlier. This heuristic did not turn out to be effective; it actually doubled the execution time instead of making it faster.

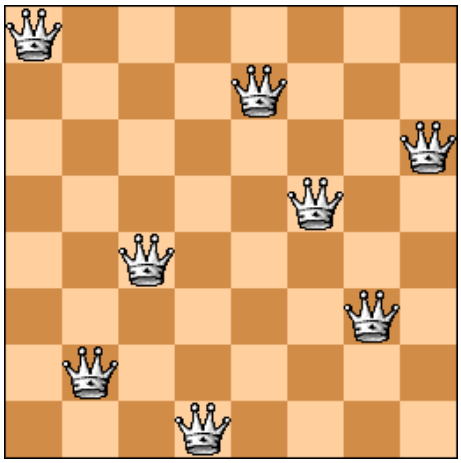

Figure 8. A solution to the eight queens problem (drawn with `www.chessup.net`)

The postage stamp problem is a challenging problem, where the complexity appears to increase exponentially with respect to the number of available denominations. Shallit [48] has shown that the related problem of finding the smallest postage which cannot be paid using at most h stamps with values drawn from a given set of denominations $\{a_1, a_2, \ldots, a_k\}$ is NP-hard, but can be solved in polynomial time if the number of denominations k is fixed.

4. Arrangement puzzles

In this section we present XQuery solutions to various arrangement puzzles, which are the n queens problem (Sec. 4.1), "Instant Insanity" (Sec. 4.2), "Tricky Triangles" (Sec. 4.3), and Sudokus (Sec. 4.4).

4.1. The n queens problem

Solutions to many combinatorial search problems are natural to express as nondeterministic algorithms. Robert Floyd presented in his seminal paper [22] a systematic way to convert nondeterministic search algorithms into backtracking programs. Floyd used the eight queens problem as an example of his approach. The *eight queens problem* is concerned with placing eight chess queens on a standard 8×8 board so that none of them is attacked by any other. An example of an arrangement which is a solution to this problem is shown in Fig. 8.

We present an XQuery solution to the *n queens problem*, which is a generalization of the eight queens problem to place n queens in a non-attacking arrangement on an $n \times n$ chess board. For $n = 1$ the problem has a trivial solution, while for $n = 2$ and $n = 3$ there are none.

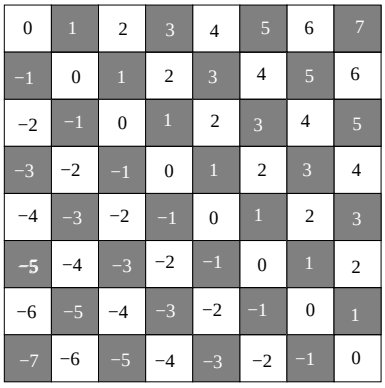

Figure 9. Diagonals of a chess board

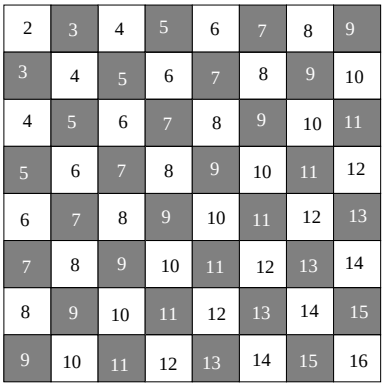

Figure 10. Counter-diagonals of a chess board

For $n = 4$ our program displays the two solutions below (which are mirror images of each other):

```
---------          ---------
| |Q| | |          | | |Q| |
---------          ---------
| | | |Q|          |Q| | | |
---------    and   ---------
|Q| | | |          | | | |Q|
---------          ---------
| | |Q| |          | |Q| | |
---------          ---------
```

(1)

The case $n = 8$ is the ordinary eight queens problem, which has 92 solutions (out of which 12 are unique modulo symmetry). The arrangement shown in Fig. 8 is the first of the solutions in the order generated by our program.

We present a straight-forward implementation of Floyd's algorithm [22] extended for arbitrary n. (As a cosmetic difference, Floyd places the queens on suitable rows in a column-wise order, while we place them on suitable columns in a row-wise order.) How to enforce the constraint that two queens do not attack each other is a key decision in the design of the algorithm. Placing the queens initially on different rows is a good start. (This is an instance of a general technique called *preclusion* in [6]. The point is to exclude impossible solutions as soon as possible.) In addition we need to control that the queens are not placed on the same column and that they do not attack each other diagonally. For the latter condition, we number the diagonals and the counter-diagonals of a board. A *diagonal* is formed of those squares which have the same difference between their row and column index. A *counter-diagonal* consists of those squares for which the *sum* of their row and column indices is the same. A standard chess board has 15 diagonals, which can be identified by differences of column and row indices using numbers $-7, -6, \ldots, 6, 7$; see Fig. 9. Similarly there are 15 counter-diagonals, which can be identified by sums of the column and row indices using numbers $2, 3, \ldots, 16$; see Fig. 10.

```
1   declare function Q:solve($usedCols as xs:integer+,
2                            $usedDiags as xs:integer+,
3                            $usedCntrDiags as xs:integer+) as xs:string* {
4     if (count($usedCols) = $N) then (: all queens have been placed :)
5           Q:displayBoard($usedCols)
6     else ( (: place a queen on the next row :)
7       let $row := count($usedCols) + 1
8       for $col in (1 to $N)
9       where not ($col = $usedCols) and
10            not ($col - $row = $usedDiags) and
11            not ($col + $row = $usedCntrDiags)
12      return Q:solve( ($usedCols, $col),
13                      ($usedDiags, $col - $row),
14                      ($usedCntrDiags, $col + $row) )  ) };
```

Figure 11. An XQuery function to solve the n queens problem

Floyd used a multiple-valued function *choice(x)*, whose values are positive integers less than or equal to x, as a main tool for designing nondeterministic algorithms. The use of *choice(x)*, and iteration, can be simulated in XQuery by a `for` clause, and recursion.

Our solution is based on the recursive function `Q:solve()` shown in Fig. 11. The function receives three sequences of integers, which contain indices of the (i) columns, (ii) diagonals, and (iii) counter-diagonals on which queens have already be placed. The function is invoked by a FLWOR expression, which tries different placements of a queen on the first row of a board:

```
for $q in (1 to $N) (: try each column of the first row :)
return Q:solve($q, $q - 1, $q + 1)
```

When all `$N` queens have been successfully placed, the function displays the solution determined by their column positions (lines 4–5). Otherwise the function tries to position a queen on the next row (line 7) at any position such that the column, the diagonal and the counter-diagonal of the queen have not been used previously. These conditions are conveniently expressed with general comparisons on lines 9–11. For any successful placement, the function recurses to find positions of queens on the remaining rows (lines 12–14). For this, the column, the diagonal and the counter-diagonal of the newly positioned queen are appended to the corresponding parameters `$usedCols`, `$usedDiags` and `$usedCntrDiags`.

Each satisfying arrangement is displayed using the function `Q:displayBoard()` as a single string similar to those in (1) above. The function receives the column indices of queens on rows 1, . . . , `$N` of the board. The display routine is a straight-forward application of useful XQuery programming techniques. For this reason—and for completeness—the remaining declarations of our n queens program are shown in Fig. 12. The built-in function `string-join()` is useful for concatenating the strings of a given sequence into a single string. The second parameter of `string-join()` specifies a string which is used as a separator between the concatenated strings. We use the empty string for this purpose (on lines 14 and 20 of Fig. 12).

```
1   declare namespace Q = "http://www.uef.fi/cs/XQueryTesting/queens";
2   declare option saxon:output "omit-xml-declaration=yes";
3
4   declare variable $N external;
5
6   (: Constants for formatting the display: :)
7   declare variable $LF := "&#10;";                  (: linefeed :)
8   declare variable $rowStart := concat($LF, "  "); (: LF + indent :)
9
10  (: Display the queens at $columns of rows 1, 2, ..., $N :)
11  declare function Q:displayBoard($columns as xs:integer+) as xs:string
12  { string-join(( $rowStart, Q:rule($N),
13                  for $col in $columns
14                  return Q:displayRow($col), $LF ), "") };
15
16  declare function Q:displayRow($col as xs:integer) as xs:string {
17    string-join( ($rowStart,
18                  for $i in (1 to $N)
19                  return if ($i eq $col) then "|Q" else "| ",
20                  "|", $rowStart, Q:rule($N) ), "") };
21
22  (: Create an $n columns wide horizontal rule :)
23  declare function Q:rule($n as xs:integer) as xs:string {
24    if ($n eq 1) then "---"
25    else concat("--", Q:rule($n - 1)) };
```

Figure 12. Auxiliary declarations for displaying positions of queens

The number of solutions to the n queens problem, denoted by $a(n)$, grows quickly as a function of n. For $n = 13$ and $n = 14$ it is respectively 73,712 and 365,596, and $a(20)$ exceeds 39×10^9. No exact formula is known for $a(n)$, but it surely grows super-polynomially; $a(n)$ is conjectured to be asymptotic to $n!/c^n$ with c around 2.54 [41].

Run times and the memory usage of our n queens program are plotted for values $n = 1, 2, \ldots, 13$ in Fig. 13 and in Fig. 14. Notice the use of logarithmic scale on the y-axis. The time usage ranges from less than a second to about one minute. Generating the 365,596 solutions to the 14 queens problem (not shown in Fig. 13) took about 6 min. The memory usage of the executions shown in Fig. 14 ranges from roughly 30×10^6 B to roughly 343×10^6 B.

4.2. Instant Insanity

The puzzle known as *Instant Insanity* (trademark of Parker Brothers) consists of four cubes, the sides of which are colored by four different colors; see Fig. 15. The task is to build a tower such that each color occurs exactly once on each of its four sides. The ordering of the cubes in the tower does not matter. The only constraint is that the visible faces of the cubes do not show any color twice on any side of the tower.

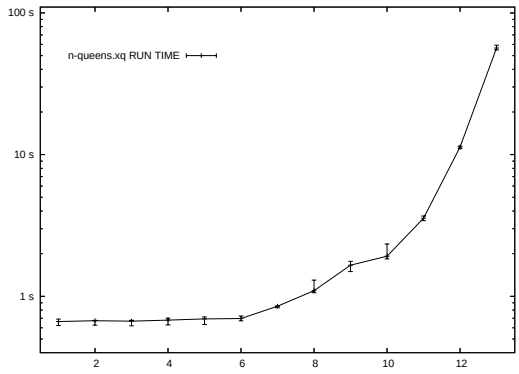

Figure 13. Execution time and ...

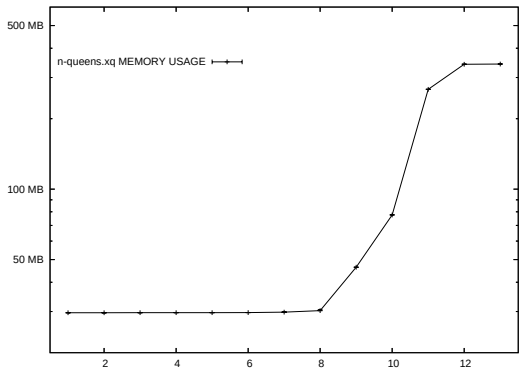

Figure 14. memory usage of the n queens program

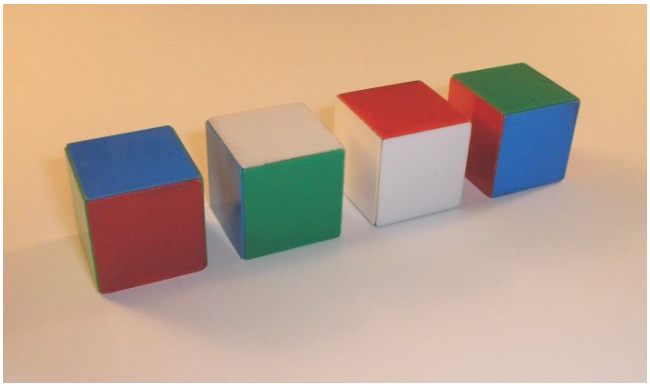

Figure 15. The "Instant Insanity" puzzle (source: Wikimedia Commons).

There are 41,472 different ways of stacking four cubes as a tower, modulo turning the tower around or upside down. The bottom cube can be placed in three different ways, by choosing which pair of opposing faces becomes hidden, one face at the top and the other at the bottom. Each of the remaining cubes have 24 different rotations, which gives gives $3 \times 24^3 = 41,472$ as total. This is a big number of combinations to be tested for a human, but not for a computer, which makes "Instant Insanity" an optimal target for a brute-force search.

We describe an ad-hoc XQuery solution for a specific instance of Instant Insanity. The solution required a fair amount of manual coding to describre the blocks and their possible rotations. The full length of the program is about 140 lines. For this reason we do not present its full code, but explain the main ideas of its various parts instead.

The program consists of a FLWOR expression, which first introduces variables for the blocks, then iterates over their rotations, and finally returns as solutions those combinations of rotations which do not use the same color on any side of the tower twice. Empty `block` elements with six attributes `back`, `right`, `front`, `left`, `bottom`, and `top` were chosen as a representation of blocks and the colors of their faces. The below `let` clause declares variables for the blocks, which are used in the rest of the FLWOR expression:

```
let $block1 := <block back="white" right="green" front="yellow"
                      left="green" bottom="yellow" top="red" />,
    $block2 := <block back="red" right="yellow" front="red"
                      left="white" bottom="red" top="green" />,
    $block3 := <block back="green" right="white" front="red"
                      left="yellow" bottom="white" top="green" />,
    $block4 := <block back="yellow" right="red" front="red"
                       left="white" bottom="white" top="green" />
```

Next a `for` clause introduces variables which each range over the rotations of the corresponding block:

```
for $R1 in blocks:Placements($block1),
    $R2 in blocks:Rotations($block2),
    $R3 in blocks:Rotations($block3),
    $R4 in blocks:Rotations($block4)
```

The rotations were defined with two different functions `Placements()` and `Rotations()`. The first was applied to the block which was chosen to be at the bottom. Elimination of identical solutions is one of the general backtracking programming techniques recognized and recommended by Bitner and Reingold [6]. The general principle of such *branch merging* is, when possible, to avoid searching branches of the search space which are isomorphic to branches that have already been searched. In order to avoid solutions which are identical modulo turning the tower around or upside down, the `Placements()` function returns only three different rotations determined by the pair of opposite faces of the cube which are are not visible, that is, one at the top and the other at the bottom. These rotations were realized as a sequence of three `block` elements generated from a given input element `$block` as follows:

```
declare function blocks:Placements($block as element(block)) as element(block)+
{      <block top="{$block/@top}" bottom="{$block/@bottom}"
           left="{$block/@left}" right="{$block/@right}"
           front="{$block/@front}" back="{$block/@back}" />,
        <block top="{$block/@left}" bottom="{$block/@right}"
           left="{$block/@bottom}" right="{$block/@top}"
           front="{$block/@front}" back="{$block/@back}" />,
```

```
         <block top="{$block/@front}" bottom="{$block/@back}"
            left="{$block/@left}" right="{$block/@right}"
            front="{$block/@bottom}" back="{$block/@top}" />  };
```

Notice how location path expressions are applied to the parameter `$block` to insert its colors in the attributes of the constructed elements.

The function `Rotations()`, which is used to generate the 24 possible rotations for each of the three remaining blocks was the most tedious part of the program to write. It simply lists the four possible rotations, for each choice of the six sides of a block facing up, as follows:

```
declare function blocks:Rotations($block as element(block)) as element(block)+
{    (: 'top' facing up: :)
        <block top="{$block/@top}" bottom="{$block/@bottom}"
           left="{$block/@left}" right="{$block/@right}"
           front="{$block/@front}" back="{$block/@back}" />,
        <block top="{$block/@top}" bottom="{$block/@bottom}"
           left="{$block/@front}" right="{$block/@back}"
           front="{$block/@right}" back="{$block/@left}" />,
        <block top="{$block/@top}" bottom="{$block/@bottom}"
           left="{$block/@right}" right="{$block/@left}"
           front="{$block/@back}" back="{$block/@front}" />,
        <block top="{$block/@top}" bottom="{$block/@bottom}"
           left="{$block/@back}" right="{$block/@front}"
           front="{$block/@left}" back="{$block/@right}" />,
  (: ... 16 SIMILAR ROTATIONS EXCLUDED ... :)
     (: 'back' facing up: :)
        <block top="{$block/@back}" bottom="{$block/@front}"
           left="{$block/@left}" right="{$block/@right}"
           front="{$block/@top}" back="{$block/@bottom}" />,
        <block top="{$block/@back}" bottom="{$block/@front}"
           left="{$block/@top}" right="{$block/@bottom}"
           front="{$block/@right}" back="{$block/@left}" />,
        <block top="{$block/@back}" bottom="{$block/@front}"
           left="{$block/@right}" right="{$block/@left}"
           front="{$block/@bottom}" back="{$block/@top}" />,
        <block top="{$block/@back}" bottom="{$block/@front}"
           left="{$block/@bottom}" right="{$block/@top}"
           front="{$block/@left}" back="{$block/@right}" />  };
```

After having introduced different block rotations for variables `$R1`, `$R2`, `$R3`, and `$R4` using the above functions, those of their combinations that were compatible with each other were returned as solutions, as follows:

```
where blocks:RotationsCompatible(($R1, $R2, $R3, $R4))
return <solution>{$R1, $R2, $R3, $R4}</solution>
```

The compatibility of the rotated blocks was tested with the function `RotationsCompatible()`. This was done by checking that each block differed from the preceding ones by the color of each of its visible faces, as follows:

```
declare function blocks:RotationsCompatible($blocks as element(block)+)
        as xs:boolean {
  every $i in 2 to count($blocks) satisfies
```

```
      not($blocks[$i]/@left = $blocks[position() lt $i]/@left) and
      not($blocks[$i]/@right = $blocks[position() lt $i]/@right) and
      not($blocks[$i]/@front = $blocks[position() lt $i]/@front) and
      not($blocks[$i]/@back = $blocks[position() lt $i]/@back) };
```

A quantified expression (`every $i in ... satisfies ...`) and the use of general comparisons to test the attributes of the ith block against all preceding blocks provided an intuitive way to express the condition of rotation compatibility.

Altogether, the FLWOR expression described above found a single solution shown below:

```
<solution>
  <block bottom="green" back="white" front="yellow"
         left="yellow" right="red" top="green"/>
  <block bottom="red" back="red" front="green"
         left="white" right="yellow" top="red"/>
  <block bottom="yellow" back="green" front="white"
         left="red" right="green" top="white"/>
  <block bottom="white" back="yellow" front="red"
         left="green" right="white" top="red"/>
</solution>
```

The evaluation of this program took about 2.3 s. The program was also optimized by replacing the call of the `RotationsCompatible()` function by a conjunction of 12 specialized conditions which did not use quantification (`every-in`). The optimized conditions use value comparisons between the attributes of the first and the second block, and general comparisons to compare the attributes of the third and of the fourth block against their predecessors. The resulting program is less elegant, but on the other hand it runs clearly faster, in about 1.2 s. With larger instances it would be reasonable to extend partial solutions gradually, similarly to Sections 4.1 and 4.3, by generating only rotations which are compatible with previous ones, instead of the above "generate-and-test-all" approach.

A generalization of Instant Insanity with arbitrarily many cubes and equally many colors is known to be NP-hard [27]. Finding a solution to the above 4-cube puzzle manually, by playing with a set of cubes, is not too tedious; actually it is quite fun. On the other hand, checking that there are *no other* solutions seems reasonable to do programmatically instead of manually. Below we have an example of a puzzle which has indeed two unique solutions.

4.3. Tricky Triangles

Next we present an XQuery solution to a "Tricky Triangles" puzzle (trademark of Dan Gilbert Art Group). A Tricky Triangles puzzle consists of nine triangular cards decorated with figures, say, of animals; see Fig. 16. The task is to arrange the cards into a 3×3 triangle so that the figures at the joining edges match. The cards of the specific puzzle considered here contain figures of five different kinds of dolphins: a dark blue, a light blue, a white, and two green ones, big and small. To make it easy to test which figures match, their parts were encoded as opposite integers, as shown in Table I. Notice that the puzzle shown in Fig. 16 is assembled incorrectly: the second row contains a mismatch between the rear of the light-blue dolphin (4) and the tail of a dark-blue dolphin (-2).

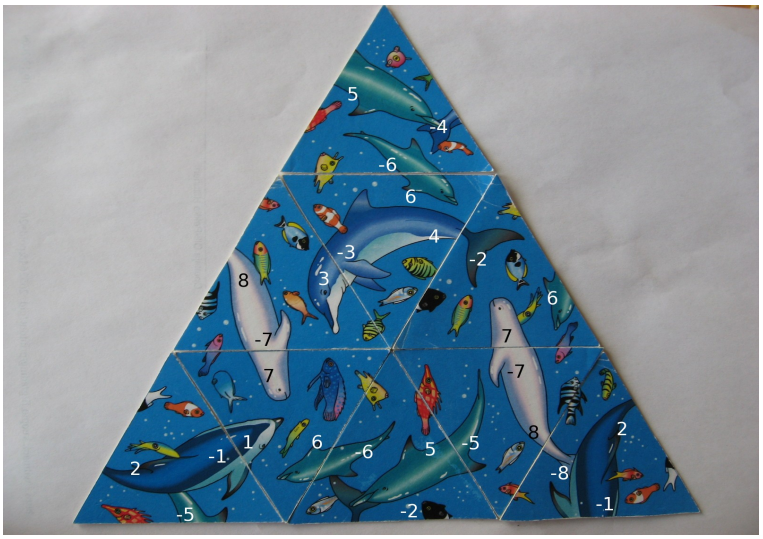

Figure 16. A "Tricky Triangles" puzzle (published with Dan Gilbert's permission). Numbers used for encoding figures have been added on top of the original pieces.

Table I. The encoding used for parts of dolphin figures

Dolphin	Head	Front	Rear	Tail
Dark blue	1	-1	2	-2
Light blue	3	-3	4	-4
Green, big	NA	5	-5	NA
Green, small	6	-6	NA	NA
White	7	-7	8	-8

Our XQuery solution represents the cards by empty `card` elements. Parts of the figures are represented by attributes `side1`, `side2` and `side3` in a clock-wise order of card edges. For example, the top-most card of Fig. 16, and the one below it, are represented by the elements `<card side1="-4" side2="-6" side3="5"/>` and `<card side1="6" side2="4" side3="-3"/>`. The nine cards are introduced to the program as the value of the below global variable:

```
declare variable $cards := (
  <card side1="1"  side2="7"  side3="6"/>, <card side1="7"  side2="-2" side3="6"/>,
  <card side1="-4" side2="-6" side3="5"/>, <card side1="-1" side2="-5" side3="2"/>,
  <card side1="-2" side2="-6" side3="5"/>, <card side1="-1" side2="-8" side3="2"/>,
  <card side1="4"  side2="-3" side3="6"/>, <card side1="8"  side2="3"  side3="-7"/>,
  <card side1="8"  side2="-5" side3="-7"/>  );
```

The solution needs to consider rotations of cards, which are generated using the below `rotations()` function. The function returns a new `card` element, which represents the three rotations of its input `$card` by its child elements of type `turn`:

```
declare function tr:rotations($card as element(card)) as element(card) {
  <card>
    <turn side1="{$card/@side1}" side2="{$card/@side2}" side3="{$card/@side3}"/>
    <turn side1="{$card/@side3}" side2="{$card/@side1}" side3="{$card/@side2}"/>
    <turn side1="{$card/@side2}" side2="{$card/@side3}" side3="{$card/@side1}"/>
  </card> };
```

Based on the above constructs, the program solves the puzzle by generating satisfying arrangements of rotated cards using a `solutions()` function, whose code is given in Fig. 17. The function is invoked as follows:

```
1   let $cardsRotated := $cards/tr:rotations(.)
2   return tr:solutions($cardsRotated)
```

XPath 2.0 and XQuery allow arbitrary expressions as location path steps, including user-defined functions. This convenient feature is applied above on line 1 to replace each of the empty `card` elements by a non-empty one which represents rotations by its child elements.

The solution is based on placing cards (possibly rotated) in the nine slots of the big triangle, starting from top. Subsequent slots are filled in an order which allows the next card to be chosen from among those that match an edge of a previously placed card; see, e.g., lines 8 and 11 in Fig. 17. The role of the variables used by function `solutions()` as place-holders of the slots is shown in Fig. 18. When a turn of a card has been selected (e.g., on line 4), that card is excluded from further consideration (e.g., on line 5). This is conveniently expressed as a node-sequence difference (`except`) between `$cards` and the parent `card` element of the chosen `turn` element. The chosen `turn` element retains its identity as a part of the XML structure, which allows its parent element be accessed via the XPath *parent axis*.

An assignment statement can be simulated within a FLWOR expression using `let` clauses; see lines 5, 9, 12, etc. Technically each of these `let` clauses introduces a *new* variable `$cards` with a new scope. That is, occurrences of `$cards` in the right-hand side expression of a `let` clause refer to its previous value, while occurrences of `$cards` in subsequent clauses of the FLWOR expression refer to the new variable declared by this `let` clause.

Tricky Triangles give raise to surprisingly many ways of arranging cards. Any of the nine cards can be placed in the first slot as three rotations. The next slot can be filled by any of the remaining eight cards, each again as three rotations, and so fourth. Thus there are in total

$$(3 \cdot 9) \cdot (3 \cdot 8) \cdots (3 \cdot 1) = 3^9 \cdot 9! = 7,142,567,040$$

different arrangements. This number counts rotations of the entire puzzle as different arrangements. Disregarding them, the number of unique arrangements is "only" $7,142,567,040/3 = 2,380,855,680$. The strategy of placing cards one-by-one in slots whose neighboring cards restrict the number of matching possibilities was quite effective for reducing the search. The number of possible values that are examined for the ith variable, for $i = 1, \ldots, 9$, is shown in Table II both for a brute-force strategy that generates all

```
1  declare function tr:solutions($cards as element(card)+) as element(solution)* {
2
3    (: Start from the top card, $c1: :)
4    for $c1 in $cards/turn
5    let $cards := $cards except $c1/parent::card
6
7    (: Then try matching rotations for the middle card of the second row: :)
8    for $c3 in $cards/turn[@side1 = -$c1/@side2]
9    let $cards := $cards except $c3/parent::card
10
11   for $c2 in $cards/turn[@side1 = -$c3/@side3]
12   let $cards := $cards except $c2/parent::card
13
14   for $c4 in $cards/turn[@side3 = -$c3/@side2]
15   let $cards := $cards except $c4/parent::card
16
17   for $c6 in $cards/turn[@side1 = -$c2/@side2]
18   let $cards := $cards except $c6/parent::card
19
20   for $c5 in $cards/turn[@side1 = -$c6/@side3]
21   let $cards := $cards except $c5/parent::card
22
23   for $c7 in $cards/turn[@side3 = -$c6/@side2]
24   let $cards := $cards except $c7/parent::card
25
26   for $c8 in $cards/turn[@side1 = -$c4/@side2 and
27                          @side3 = -$c7/@side1]
28   let $cards := $cards except $c8/parent::card
29
30   for $c9 in $cards/turn[@side3 = -$c8/@side2]
31   return <solution>{$c1, $c2, $c3, $c4, $c5, $c6, $c7, $c8, $c9}</solution> };
```

Figure 17. An XQuery function for solving "Tricky Triangles"

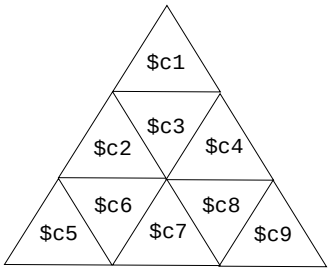

Figure 18. The role of variables used for solving "Tricky Triangles"

Table II. Number of possible assignments to different card positions

Strategy	1st	2nd	3rd	4th	5th	6th	7th	8th	9th
all-inclusive	27	648	13,608	244,944	$4 \cdot 10^6$	$4 \cdot 10^7$	$4 \cdot 10^8$	$2 \cdot 10^9$	$7 \cdot 10^9$
`tr:solutions()`	27	48	81	129	138	135	123	12	6

possibilities without considering the matching of figures, and the strategy used by function `tr:solutions()`. (Numbers above a million are approximate values in the table.)

Bitner and Reingold have mentioned this kind of *search rearrangement* as one of the general backtrack programming techniques, and given it the following description [6]:

> In general, nodes of low degree should occur early in the search tree, and nodes of high degree should occur later. Since preclusion frequently occurs at a fixed depth, fewer nodes may need to be examined. When faced with the choice of several ways of extending the partial solution (i.e. which square to tile next or in which column to place the next queen [...]), we choose the one that offers the fewest alternatives.

Similar search rearrangement for solving Sudoku puzzles is considered in Section 4.4.

The described XQuery program is quite efficient; it uses about 0.8 s of execution time and about 30 MB of main memory to find two unique solutions to the puzzle, which are represented by the below elements:

```
<solution>
  <turn side1="6"  side2="1"  side3="7" />
  <turn side1="-2" side2="-6" side3="5" />
  <turn side1="-1" side2="-8" side3="2" />
  <turn side1="-5" side2="-7" side3="8" />
  <turn side1="3"  side2="-7" side3="8" />
  <turn side1="6"  side2="4"  side3="-3" />
  <turn side1="-6" side2="5"  side3="-4" />
  <turn side1="7"  side2="-2" side3="6" />
  <turn side1="-1" side2="-5" side3="2" />
</solution>
<solution>
  <turn side1="-1" side2="-8" side3="2" />
  <turn side1="7"  side2="6"  side3="1" />
  <turn side1="8"  side2="-5" side3="-7" />
  <turn side1="-2" side2="-6" side3="5" />
  <turn side1="4"  side2="-3" side3="6" />
  <turn side1="-6" side2="5"  side3="-4" />
  <turn side1="2"  side2="-1" side3="-5" />
  <turn side1="6"  side2="7"  side3="-2" />
  <turn side1="8"  side2="3"  side3="-7" />
</solution>
```

The program produces actually *six* different solutions, by presenting the above unique solutions as three rotations (of the entire puzzle). There does not seem to be an obvious trick to avoid

1					7		9	
	3			2				8
		9	6			5		
		5	3			9		
	1			8				2
6					4			
3							1	
	4							7
		7				3		

```
<board>
  <row>1,0,0,0,0,7,0,9,0</row>
  <row>0,3,0,0,2,0,0,0,8</row>
  <row>0,0,9,6,0,0,5,0,0</row>
  <row>0,0,5,3,0,0,9,0,0</row>
  <row>0,1,0,0,8,0,0,0,2</row>
  <row>6,0,0,0,0,4,0,0,0</row>
  <row>3,0,0,0,0,0,0,1,0</row>
  <row>0,4,0,0,0,0,0,0,7</row>
  <row>0,0,7,0,0,0,3,0,0</row>
</board>
```

Figure 19. The "AI Escargot" Sudoku [29] (left; published with Arto Inkala's permission) and its representation as an XML document (right)

this. If the puzzle had a slot at the center of symmetry, identical solutions could be avoided by placing each card in the center slot as one rotation only. This approach works on some similar puzzles, like the "Ferocious Big Cats Nature Puzzle" (trademark of Lagoon Puzzles). The latter puzzle consists of nine square tiles, instead of triangles, decorated with images of tigers, lions, and panthers, which are to be arranged as a matching 3×3 square.

4.4. Sudoku puzzles

A *Sudoku* puzzle consists of a 9×9 grid of cells, some of which have been filled with numbers from 1 to 9; see Fig. 19. The task is to fill the remaining cells with numbers from 1 to 9 so that each number appears exactly once in each row, in each column, and in each of the nine 3×3 sub-grids. (We call these sub-grids "boxes", and the entire 9×9 grid a "board"). Sudoku puzzles are published in newspapers and in dedicated specialty magazines world-wide. They have become so popular that computer security expert Ben Laurie has stated, according to Norvig [40], that Sudoku is "a denial of service attack on human intellect".

We describe a straight-forward XQuery program for solving Sudoku puzzles. The program uses empty `cell` elements with attributes `rowNum`, `colNum`, `val`, and `box` to represent the contents of the cells. The contents of empty cells is represented with `val="0"`. For example, the sixth and the seventh cell of the first row in Fig. 19 are represented by the below elements:

```
<cell rowNum="1" colNum="6" val="7" box="2" />
<cell rowNum="1" colNum="7" val="0" box="3" />
```

The XML format of Sudoku boards accepted as input by our program is shown in Fig. 19. When the parameter `$SudoDoc` has been bound to the URI of such an input document, the contents of the board are first transformed into a sequence of `cell` elements as follows:

```
let $cells := sudo:preprocess(doc($SudoDoc)//row)
```

The `preprocess()` function is relatively straight-forward; its declarations are shown in Fig. 20. The function iterates over the rows of the input board, and their numbers (line 6). The contents

```
1   declare namespace sudo="http://www.uef.fi/cs/XQueryTesting/sudoku";
2   declare variable $SudoDoc external;
3
4   (: Return the cells of the rows adorned with row, column and box numbers :)
5   declare function sudo:preprocess($rows as element(row)+) as element(cell)+ {
6     for $row at $rowNum in $rows
7     let $colContents := tokenize(string($row), ",\s*")
8     for $colCont at $colNum in $colContents
9     return <cell rowNum="{$rowNum}" colNum="{$colNum}"
10              val="{xs:integer($colCont)}" box="{sudo:boxNum($rowNum, $colNum)}" /> };
11
12  declare function sudo:boxNum($rowNum as xs:integer, $colNum as xs:integer)
13          as xs:integer { (: box number in 1,2,...,9 :)
14    (($rowNum - 1) idiv 3)*3 + ($colNum - 1) idiv 3 + 1 };
```

Figure 20. Declarations for loading a Sudoku board as an XML document

of each row is split into a sequence, `$colContents`, of strings which are separated by a comma followed by zero or more white-space characters, using the built-in function `tokenize()` on line 7. Finally, each of these content strings with its row and column number are used to create a corresponding `cell` element (lines 9–10). The expression `xs:integer($colCont)` is a *type constructor*, which casts its argument value to an integer. This is not necessary for the solution, but it provides an easy trick to get rid of leading or trailing white-space that could remain at the start of the first cell or at the end of the last cell as the result of splitting the contents of a row. The `idiv` operator used on line 14 stands for truncating integer division.

After input preprocessing, the puzzle is solved with a simple recursive `solution()` function. The function takes two input sequences. The first of them consists of the cells which have not been assigned a value yet, and the other of the cells which have been assigned a value. The `solution()` function is shown in Fig. 21, and it is invoked as follows:

```
let $freeCells := $cells[@val = 0],
    $fixedCells := $cells except $freeCells
for $solution in sudo:solution($freeCells, $fixedCells)
return sudo:displayCells($solution/*)
```

The `displayCells()` function is a straight-forward routine for displaying the values of the cells, which comprise the contents of a board, as a grid. For completeness also the code of the `displayCells()` function is shown in Fig. 22.

When all cells have been filled, the `solution()` function returns the solution as a `board` element which contains the filled cells (lines 3–4 in Fig. 21). Otherwise the function picks the first of the unfilled cells and tries to fill it with numbers not already used in the same row, cell or box (lines 6–11). The bar '`|`' is a node sequence union operator. So the expression `($thisRow | $thisCol | $thisBox)/@val` evaluates to the values of any cell that appears in the same row, column or box with the chosen `$cell`. For each digit which is not among

```
1   declare function sudo:solution($freeCells as element(cell)*,
2                                  $fixedCells as element(cell)+) as element(board)*
3   { if ( empty($freeCells) ) then (: the board is complete :)
4       <board>{$fixedCells}</board>
5     else
6       let $cell := $freeCells[1] (: pick the first unfilled cell :)
7       let $thisRow := $fixedCells[@rowNum eq $cell/@rowNum],
8           $thisCol := $fixedCells[@colNum eq $cell/@colNum],
9           $thisBox := $fixedCells[@box eq $cell/@box]
10      let $forbiddenVals := ($thisRow | $thisCol | $thisBox)/@val
11      for $val in (1 to 9)[not(. = $forbiddenVals)]
12      let $fixedCells2 :=
13            ($fixedCells, <cell val="{$val}">{ $cell/@*[name() ne "val"] }</cell>)
14      return sudo:solution($freeCells except $cell, $fixedCells2)  };
```

Figure 21. The `solution()` function for solving Sudoku boards

```
1   declare variable $line-btw-boxes := "---------------------&#10;";
2
3   declare function sudo:displayCells($cells as element(cell)+) as xs:string+
4   { for $rowNum in (1 to 9)
5     return ( sudo:displayRow($cells[@rowNum = $rowNum]),
6              if ($rowNum eq 3 or $rowNum eq 6) then $line-btw-boxes
7              else () ) };
8
9   declare function sudo:displayRow($cells as element(cell)+) as xs:string+
10  { for $colNum in (1 to 9)
11    return ($cells[@colNum = $colNum]/@val, (: + add a bar btw boxes: :)
12            if ($colNum eq 3 or $colNum eq 6) then "|" else () ), "&#10;" };
```

Figure 22. Declarations for displaying Sudoku grids

these forbidden values, the function is invoked recursively, the chosen cell with its assigned value added to the sequence of the filled cells (lines 12–13), and removed from the sequence of the unfilled cells (line 14). The function `name()` invoked without an argument evaluates to the name of the context node. So the expression `$cell/@*[name() ne "val"]` on line 13 copies the attributes of the element `$cell`, except the `val` attribute, which is assigned a new value by the element constructor.

Because the `solution()` function places the cells as the children of its result `board` element in an irregular order, the display routine processes the rows (lines 4–5 in Fig. 22) and their columns (lines 10–11) explicitly in the order of their numbers.

Human solvers seldom resort to brute-force search similar to the `solution()` function. Rather, they apply sophisticated strategies for shortening the deduction of the missing values

```
1   (: Return unfilled cells which have maximally many values in their row, column and box :)
2   declare function sudo:mostConstrainedFreeCells(
3           $freeCells as element(cell)+, $fixedCells as element(cell)+) as element(cell)* {
4       let $maxNumOfConstraints := max(for $cell in $freeCells
5                                       return sudo:numOfConstraints($cell, $fixedCells))
6       for $cell in $freeCells
7       where sudo:numOfConstraints($cell, $fixedCells) eq $maxNumOfConstraints
8       return $cell };
9
10  declare function sudo:numOfConstraints(
11          $cell as element(cell), $fixedCells as element(cell)+) as xs:integer {
12      let $neighbors := $fixedCells[@rowNum eq $cell/@rowNum or
13                                    @colNum eq $cell/@colNum or
14                                    @box eq $cell/@box]
15      return count(distinct-values($neighbors/@val)) };
```

Figure 23. Functions for greedy selection of cells by the number of their constraints

of cells; see, e.g., [15]. No advanced strategies were considered in our XQuery implementation, except for a straight-forward search rearrangement by choosing the next cell to be filled from among those which have the smallest number of possible assignments left. This greedy strategy was realized by a function `mostConstrainedFreeCells()` shown in Fig. 23, which returns a sequence of those currently unfilled cells that have a maximal number of different values assigned in the neighborhood consisting of their row, column and box. The function is invoked from the `solution()` function (as a replacement of line 6 in Fig. 21) as follows:

```
(: pick some maximally constrained cell :)
let $cell := sudo:mostConstrainedFreeCells($freeCells, $fixedCells)[1]
```

An auxiliary function `numOfConstraints()` (lines 10–15 in Fig. 23) counts how many different numbers have been assigned in the row, column and box of a given cell. This function is first used to find the maximum number of constraints among any unfilled cells (lines 4–5), and then to select those of them that have this many constraints (lines 6–8).

The efficiency of the program and the effectiveness of the greedy search rearrangement varies considerably based on the puzzle instance. Results of a few sample runs are shown in Table III. The puzzles identified as *easy004* and *hard004* are the puzzles published with number 004, respectively, under the "Easy" and the "Hard" category by Livewire Puzzles[§]. *Escargot* is the puzzle shown in Fig. 19, which was published by Inkala in 2006 as "the most difficult Sudoku puzzle" [29]. *Fiendish 2* is a puzzle found with that name in [54]. *Inkala10* is another difficult puzzle by Inkala taken from [39]. The puzzles named *minimal17* and *double16* are from [17, p. 83]. The first one of them is an example of a correct puzzle with only 17 fixed cells, which

[§] `http://www.puzzles.ca/sudoku.html`

Table III. XQuery time and memory usage on some Sudoku puzzles

PUZZLE	UNOPTIMIZED		WITH SEARCH REARRANGEMENT			
	time	memory	time	(% or unopt)	memory	(% or unopt)
easy004	1.1 s	36 MB	1.2 s	120%	48 MB	133%
hard004	2.9 s	146 MB	1.6 s	55%	82 MB	56%
Escargot	4.0 s	272 MB	4.8 s	120%	344 MB	126%
Fiendish 2	6.9 s	357 MB	4.0 s	58%	344 MB	96%
Inkala10	7.1 s	356 MB	4.4 s	62%	344 MB	97%
minimal17	145.3 s	401 MB	90.3 s	62%	381 MB	95%
double16	158.5 s	411 MB	26.1 s	16%	354 MB	86%

seems to be the minimum for a puzzle with a single solution [17]. The *double16* puzzle has only 16 initially fixed cells and yields two alternative solutions.

The generalized version of Sudoku is, again, known to be a computationally hard problem. Yato and Seta have shown that finding solutions to Sudoku puzzles which use n^2 symbols on an $n^2 \times n^2$ grid is NP-hard [56].

5. Discussion

We have seen examples of several XQuery features which are convenient for solving recreational problems and puzzles. In this section we discuss the usability of XQuery as a tool for problem solving, based on these examples.

Compositionality and orthogonality of combining sequence-valued expressions facilitates the writing of programs in general. *Generation and filtering of sequences* yield easy solutions for some problems; see, e.g., Section 3.1. The *FLWOR expression*, which provides built-in support for creating and testing multiple variable bindings, is an easy means to simulate non-deterministic search; this was especially present in Section 4. Easy application of an expression to each item of a sequence, either using a FLWOR expression (e.g., line 2 of the `permutations()` function on page 13) or as a *location path* step applicable to a sequence of nodes (e.g., in the invocation of the `rotations()` function on page 29) is a convenient feature. The existential semantics of *general comparisons* provides a compact and natural expression for many conditions, and *quantified Boolean expressions* likewise. The latter were applied especially in the solution of the stamp problem (Fig. 7 of page 19), and the features were used together in the `RotationsCompatible()` function (on page 26).

In general, *XML/XPath trees* provide a generic representation of symbolic data, and *user-defined recursive functions* provide a generic means of repetition. Both were utilized in most of examples above.

XQuery has also drawbacks and limitations which we shall be aware of, too. One is that XQuery provides *no support for solution-at-a-time processing*. XQuery expressions which evaluate to a complete sequence of values cannot be used to specify infinite results in

the style of lazy functional programming or the backtracking of Prolog. Another high-level restriction is that XQuery does not comprise a real constraint solver. Solutions based on binding variables to values from a *finite set of candidates* support solving smallish instances of *integer* programming. On the other hand, many practical optimization problems can be cast and solved as instances of *linear programming* (see, e.g., [16, Chap. 7]), where the variables take values from uncountable subsets of reals.

The programs that we have seen are mainly brute-force solutions to combinatorial search problems. Since many of these problems are computationally hard, such solutions are necessarily applicable to smallish problem instances only, even when optimized with effective heuristics. Nevertheless, there is a niche for such solutions: some problem instances are too tedious to solve manually, but not too complex for computerized brute-force search. On such problem instances, straight-forward computer-based solutions, like those above, can be valuable help for human problem solving.

The *lack of an assignment statement* may be the biggest inconvenience for programmers accustomed with procedural languages. The lack of an assignment statement can largely be circumvented by the conventions of functional programming, that is, by passing the changed values as parameters to recursive procedures. This style of programming places challenges on efficient implementations. For example, the Saxon processor would benefit of a better implementation of tail recursion. When the depth of recursion exceeds a few thousand invocations, Saxon may either report a run-time error or throw a `StackOverflowError` exception. (This limitation may be inherited from Java [50] through its use as Saxon's implementation language.)

A fundamental restriction of XQuery variables is that they are purely a query-level concept. Variables can be used as a reference to the value that has been bound to them, but they cannot be manipulated or passed around uninstantiated, for example in a sequence or as a part of a tree structure. This is in contrast with, say, *incomplete data structures*, which are a powerful and efficient Prolog programming technique [49, Chap. 15].

How well would XQuery rate as a problem solving language? A "problem-solving language" is not a precisely defined concept. Problem solving has traditionally been considered as a subfield of Artificial Intelligence (AI). Comparing features of XQuery against features considered useful for writing AI applications provides one way of reflecting on XQuery's problem-solving capabilities. We will consider below features of AI languages as of early 80's, which Elaine Rich has analyzed in her textbook [46, Sec. 12.1].

Many of the desirable features of AI languages discussed by Rich are found also in XQuery. XQuery sequences provide *good facilities for manipulating lists*, with some reservations on the lack of nested sequences. The XML Schema type system employed by XQuery provides *a variety of data types* to describe the many kinds of information a large system needs. For example, a rich collection of built-in operations for manipulating dates, times and durations are available (not considered in this article). XML/XPath trees provide *facilities for building complex knowledge structures*. XML elements can be viewed as generic record structures, which allow arbitrary repetition and nesting of sub-structures, and flexible access via location path expressions. *Ability to decompose the system* into manageable units is supported in XQuery by user-defined functions. Functions and related declarations can also be grouped inside *library modules*, which support the development of larger XQuery applications. (Implementation of

library modules is an optional feature of XQuery.) Recursion and FLWOR expressions provide *flexible control structures*, but on the other hand several familiar control structures are excluded from XQuery. *Late binding* for such things as the size of a data structure or the type of an object to be operated on are useful in many AI programs. XQuery supports late binding via elements and sequences whose size does not need to be known in advance. XQuery also contains `instance-of` and `typeswitch` expressions, which allow different branches of the program to be evaluated based on the types of expression values.

The rest of Rich's requirements are not that well satisfied by XQuery. Sufficient *efficiency* is a relative notion, which depends on system requirements. We have found the efficiency of Saxon mostly sufficient for the experiments presented in this article.

Other desirable features of AI programming languages are clearly not met by XQuery. XQuery includes no features for *interaction with the user*. *Pattern matching* to determine control is not supported either, but XPath location path expressions provide a flexible pattern-matching based mechanism for accessing contents of XML structures. XQuery does not comprise a theorem-prover and thus supports no features for guiding deduction, like *goal-directed behavior* or *attention focusing*, unless the generate-and-test capabilities of FLWOR expressions and quantified Boolean expressions are considered as a form of *automated deduction*. Finally, AI programs often need the *ability to intermix procedures and data*, which is currently not supported by XQuery. There are plans to include *higher order functions* in the next version of XQuery [19]. Higher order functions allow one to pass a function as an argument to another function and to invoke a function that has been passed as an argument.

Bitner and Reingold [6] have mentioned four generally applicable heuristics for pruning the search space of backtracking programs. These are *preclusion*, *branch merging*, *search rearrangement*, and *branch and bound*. We have encountered examples of the first three of these heuristics in the XQuery programs above. *Branch and bound* is a search space pruning heuristic applicable to *optimization problems*, using a bound function for the value of the solutions which are obtainable as completions of a partial solution; see, e.g., [35, Sec. 12.2]. We did not consider optimization problems, and thus branch and bound was not relevant to our examples. Branch and bound involves storing partial solutions in a data structure like a priority queue. It is not difficult to use XQuery sequences (as a 2-heap) to implement priority queues, but we will not explore this possibility further. It could be an interesting topic of further study to consider how far XQuery can pushed as a data structure implementation language, along the lines of functional data structures [42]. Using XQuery as a problem-solving language may already be considered by some as pushing the language beyond its limits, but we find the XQuery programs of this article to provide rather natural solutions to the considered problems.

6. Related work

This section lists additional references on the background and works related to this article.

A good discussion of the origins of XQuery and of the design considerations by the W3C XML Query working group is given by Chamberlin [10]. Fuller's on-line article [24] gives practical advice for developing XQuery applications. Bamford et al. [2] describe a number of XQuery-related projects and application scenarios. They also give an overview of XQuery

implementation techniques, and review available XQuery implementations as of 2009. Kay has discussed the design decisions of his Saxon XQuery implementation in [31]. Li, Liao, and Yang [37] discuss an XQuery implementation using lazy evaluation. Extensions to XQuery to support client-side programming have been proposed, implemented and demonstrated in [23].

Wan and Dobbie [53] have considered the data mining of association rules using XQuery. A more efficient implementation of the *Apriori* algorithm is presented by Romei and Turini [47], who describe an extension of XQuery, called XQuake, and a related system for XML data mining.

The use of toy problems and puzzles in computer science education has been discussed and advocated, for example, by Knuth [32, Ch. 10], Levitin and Papalaskari [36], and Parhami [44]. Puzzles, fun and games have some publications of their own related to computing. An interested reader is referred to Knuth's collected writings [34] or the proceedings of a triennial conference on fun with algorithms [14, 8].

Sudoku solvers have been written in several languages; 24 of them can be found in Rosetta Code[¶], which is a "programming chrestomathy" site. The idea is to present solutions to the same task in many different languages. Currently (in February 2011) there are 465 tasks for which solutions in 358 languages are given. The coverage of XQuery is quite modest: only 7 pages are found under the "XQuery" category. (For example, the "Java" category contains 328 pages, and the "Python" category 444 pages. The "Puzzles" category contains no tasks yet, but 21 pages are listed under the "Games" category.) Norvig has given a careful description of a Sudoku solver written in Python [40].

Gallopoulos, Houstis and Rice [25] discuss the concept and potential of *problem-solving environments*, which are computer systems that provide computational facilities to solve a class of problems. A case-study based comparison of specific problem-solving technologies is presented by Dhar and Ranganathan [18]. They compare and discuss integer programming vs symbol-manipulation based expert systems as tools for solving constraint satisfaction problems, using assignment of teachers to university courses as a case.

7. Conclusion

XQuery is a specialized XML query language, whose features allow convenient expression of solutions to many combinatorial search problems. We have demonstrated this by presenting XQuery solutions to several recreational problems and puzzles, and by discussing the usability of XQuery as a problem-solving language. XQuery is a relatively new language, and thus the techniques, tricks and conventions of XQuery programming are likely to mature in the close future. Improvements to the presented programs are therefore likely, and welcome, both via better algorithms and heuristics or via better XQuery programming techniques. Some attention was paid to the efficiency of the presented solutions, but the main concern was in their clarity

[¶] `http://rosettacode.org/wiki/Main_Page`

and simplicity. We hope that this has helped us to demonstrate the suitability and convenience of XQuery as an interesting tool for experimental problem solving.

REFERENCES

1. A.V. Aho, B.W. Kernighan, and P.J. Weinberger. *The AWK Programming Language*. Addison-Wesley, second edition, 1988.
2. R. Bamford, V. Borkar, M. Brantner, P.M. Fischer, D. Florescu, D. Graf, D. Kossmann, T. Kraska, D. Muresan, S. Nasoi, and M. Zacharioudakis. XQuery reloaded. In *Proceedings of VLDB'09*, Lyon, France, August 2009. ACM.
3. A. Berglund, S. Boag, D. Chamberlin, M.F. Fernández, M. Kay, J. Robie, and J. Siméon, editors. *XML Path Language (XPath) 2.0 (Second Edition)*. W3C Recommendation, December 2010.
4. A. Berglund, M. Fernández, A. Malhotra, J. Marsh, M. Nagy, and N. Walsh, editors. *XQuery 1.0 and XPath 2.0 Data Model (XDM) (Second Edition)*. December 2010. W3C Recommendation, available at `http://www.w3.org/TR/xpath-datamodel`.
5. P.V. Biron and A. Malhotra, editors. *XML Schema Part 2: Datatypes Second Edition*. October 2004. W3C Recommendation. Available at `http://www.w3.org/TR/xmlschema-2`.
6. J.R. Bitner and E.M. Reingold. Backtrack programming techniques. *Communications of the ACM*, 18(11):651–656, November 1975.
7. S. Boag, D. Chamberlin, M.F. Fernández, D. Florescu, J. Robie, and J. Siméon, editors. *XQuery 1.0: An XML Query Language (Second Edition)*. December 2010. W3C Recommendation, available at `http://www.w3.org/TR/xquery`.
8. P. Boldi and L. Gargano, editors. *Fun with Algorithms, 5th International Conference, FUN 2010, Ischia, Italy, June 2-4, 2010. Proceedings*, volume 6099 of *Lecture Notes in Computer Science*. Springer, 2010.
9. T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, and F. Yergeau, editors. *Extensible Markup Language (XML) 1.0*. W3C Recommendation, fifth edition, November 2008.
10. D. Chamberlin. Influences on the design of XQuery. In H. Katz, editor, *XQuery from the Experts: A Guide to the W3C XML Query Language*, chapter 2. Addison-Wesley, 2004.
11. D. Chamberlin, P. Fankhauser, D. Florescu, M. Marchiori, and J. Robie, editors. *XML Query Use Cases*. March 2007. W3C Working Group Note, available at `http://www.w3.org/TR/xquery-use-cases`.
12. D. Chamberlin, J. Robie, and D. Florescu. Quilt: An XML query language for heterogeneous data sources. In D. Suciu and G. Vossen, editors, *WebDB (Selected Papers)*, volume 1997 of *Lecture Notes in Computer Science*, pages 1–25. Springer, 2000.
13. W.F. Clocksin and C.S. Mellish. *Programming in Prolog, Second Edition*. Springer, 1984.
14. P. Crescenzi, G. Prencipe, and G. Pucci, editors. *Fun with Algorithms, 4th International Conference, FUN 2007, Castiglioncello, Italy, June 3-5, 2007, Proceedings*, volume 4475 of *Lecture Notes in Computer Science*. Springer, 2007.
15. J.F. Crook. A pencil-and-paper algorithm for solving Sudoku puzzles. *Notices of the AMS*, 56(4):460–468, April 2009.
16. S. Dasgupta, C. Papadimitriou, and U. Vazirani. *Algorithms*. McGraw-Hill, New York, NY, 2008.
17. J.-P. Delahaye. The science behind Sudoku. *Scientific American*, pages 80–87, June 2006.
18. V. Dhar and N. Ranganathan. Integer programming vs. expert systems: An experimental comparison. *Communications of the ACM*, 33(3):323–336, March 1990.
19. D. Engovatov and J. Robie, editors. *XQuery 3.0 Requirements*. September 2010. W3C Working Draft, available at `http://www.w3.org/TR/xquery-30-requirements`.
20. D. Eppstein. On the NP-completeness of cryptarithms. *ACM SIGACT News*, 18(3):38–40, April 1987.
21. D. Flanagan. *Java in a Nutshell*. O'Reilly Media, Sebastopol, CA, fifth edition, March 2005.
22. R. W. Floyd. Nondeterministic algorithms. *Journal of the Association for Computing Machinery*, 14(4):636.644, October 1967.
23. G. Fourny, D. Kossman, T. Kraska, M. Pilman, and D. Florescu. XQuery in the browser. In *Proceedings of SIGMOD'08*, Vancouver, BC, June 2008. ACM.
24. J. R. Fuller. Advancing with XQuery: Develop application idioms. IBM developerWorks on-line article, September 2008. Available at `http://www.ibm.com/developerworks/xml/tutorials/x-advxquery/`.
25. E. Gallopoulos, E. Houstis, and J.R. Rice. Computer as thinker/doer: Problem-solving environments for computational science. *IEEE Computational Science & Engineering*, 1:11–23, 1994.

26. M. Gardner. *Mathematical Puzzle Tales*. The Mathematical Association of America, Washington, D.C., 2000.
27. M.R. Garey and D.S. Johnson. *Computers and Intractability*. W.H. Freeman and Company, New York, NY, 1979.
28. G.H. Hardy and E.M. Wright. *An Introduction to the Theory of Numbers*. Oxford University Press, Oxford, UK, fourth edition, 1975.
29. A. Inkala. *AI Escargot – The Most Difficult Sudoku Puzzle*. Lulu.Com, 2007.
30. M. Kay, editor. *XSL Transformations (XSLT) Version 2.0*. January 2007. W3C Recommendation, available at `http://www.w3.org/TR/xstl20`.
31. M. Kay. Ten reasons why Saxon XQuery is fast. *IEEE Data Eng. Bull.*, 31(4):65–74, 2008.
32. D. E. Knuth. *Selected Papers on Computer Science*. CSLI Publications, Stanford, CA, 1996.
33. D. E. Knuth. *The Art of Computer Programming, Volume 4A: Combinatorial Algorithms, Part 1*. Addison-Wesley, 2011.
34. D. E. Knuth. *Selected Papers on Fun & Games*. CSLI Publications, Stanford, CA, 2011.
35. A. Levitin. *Introduction to The Design and Analysis of Algorithms*. Pearson Education, USA, second edition.
36. A. Levitin and M.-A. Papalaskari. Using puzzles in teaching algorithms. In J. L. Gersting, H. M. Walker, and S. Grissom, editors, *SIGCSE*, pages 292–296. ACM, 2002.
37. P. Li, H. Liao, and H. Yang. An implementation for XQuery based on lazy evaluation. In *First International Workshop on Database Technology and Applications*, pages 463–467, Hubei, China, April 2009. IEEE.
38. A. Malhotra, J. Melton, N. Walsh, and M. Kay, editors. *XQuery 1.0 and XPath 2.0 Functions and Operators (Second Edition)*. December 2010. W3C Recommendation, available at `http://www.w3.org/TR/xpath-functions`.
39. World's hardest sudoku: Can you solve Dr Arto Inkala's puzzle? On-line article in *Mirror.co.uk*, August 19, 2010.
40. P. Norvig. Solving every Sudoku puzzle. On-line essay at `http://norvig.com/sudoku.html`, 2009-2010.
41. On-Line Encyclopedia of Integer Sequences. Number of ways of placing n nonattacking queens on n x n board. `http://oeis.org/A000170`, February 2011.
42. C. Okasaki. *Purely Functional Data Structures*. Cambridge University Press, Cambridge, UK, 1998.
43. M. E. O'Neill. The genuine sieve of Eratosthenes. *Journal of Functional Programming*, 19(1):95–106, January 2009.
44. B. Parhami. A puzzle-based seminar for computer engineering freshmen. *Computer Science Education*, 18(4):261–277, December 2008.
45. G. Polya. *How to Solve It*. Princeton University Press, Princeton, NJ, expanded Princeton science library edition, 2004.
46. E. Rich. *Artificial Intelligence*. McGraw-Hill, Inc., New York, NY, 1983.
47. A. Romei and F. Turini. XML data mining. *Software - Practice and Experience*, 40(2):101–130, 2010.
48. J. Shallit. The computational complexity of the local postage stamp problem. *ACM SIGACT News*, 33(1), March 2002.
49. L. Sterling and E. Shapiro. *The Art of Prolog: Advanced Programming Techniques*. MIT Press, Cambridge, MA, 1986.
50. RFE: Tail call optimization. Bug # 4726340 in Sun Developer Network Bug Database, August 2002. Available at `http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=4726340`.
51. G. van Rossum. *Python Tutorial*. Stichting Mathematicsh Centrum, Amsterdam, Netherlands, October 1995. Release 1.3.
52. P. Walmsley. *XQuery*. O'Reilly Media, Inc., Sebastopol, CA, 2007.
53. J. W. W. Wan and G. Dobbie. Extracting association rules from XML documents using XQuery. In R. H. L. Chiang, A. H. F. Laender, and E.-P. Lim, editors, *WIDM*, pages 94–97. ACM, 2003.
54. A. Welch. Dimitre's tuned Sudoku solution. Blog entry at `http://ajwelch.blogspot.com/2006/03/dimitres-tuned-sudoku-solution.html`, March 2006.
55. Wikipedia. Constraint programming. Available at `http://en.wikipedia.org/wiki/Constraint_programming`, January 2011.
56. T. Yato and T. Seta. Complexity and completeness of finding another solution and its application to puzzles. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, E86-A:1052–1060, May 2003. Available at `http://search.ieice.org`.