Artificial Intelligence Review
Published online: November 2009, DOI 10.1007/s10462-009-9150-5
Published hardcopy: 33, 1 & 2, 2010, pp. 135-150.
The original publication is available at www.springerlink.com

Logic Programming for Combinatorial Problems

Toshinori Munakata - Roman Barták

Abstract

Combinatorial problems appear in many areas in science, engineering, biomedicine, business, and operations research. This article presents a new intelligent computing approach for solving combinatorial problems, involving permutations and combinations, by incorporating logic programming. An overview of applied combinatorial problems in various domains is given. Such computationally hard and popular combinatorial problems as the traveling salesman problem are discussed to illustrate the usefulness of the logic programming approach. Detailed discussions of implementation of combinatorial problems with time complexity analyses are presented in Prolog, the standard language of logic programming. These programs can be easily integrated into other systems to implement logic programming in combinatorics.

Keywords: Intelligent combinatorics; logic programming; permutations and combinations

---

Toshinori Munakata
Computer and Information Science Department, Cleveland State University
Cleveland, Ohio 44115, U.S.A.
munakata@cis.csuohio.edu

Roman Barták
Department of Theoretical Computer Science and Mathematical Logic, Charles University
Praha, Czech Republic
bartak@kti.mff.cuni.cz

1. Introduction

Combinatorics, or combinatorial theory, is a major mathematics branch that has extensive applications in many fields such as engineering (e.g., pattern such as image analyses, communication networks), computer science (e.g., languages, graphs, intelligent computing), natural and social sciences, biomedicine (e.g., molecular biology), operations research (e.g., transportation, scheduling), and business (Liu, 1968; Roberts, 1984; *Combinatorial Pattern Matching,* 1992-2009; Mycielski, et al., 1997). The most common scenario is that many real world problems are mathematically intractable. In these cases, combinatorics techniques are needed to count, enumerate, or represent possible solutions in the process of solving application problems. Generation of combinatorial sequences, such as permutations and combinations, has been studied extensively because of the fundamental nature and the importance in practical applications. There has been interest in the generation of these sequences in a parallel or distributed computing environment (Akl, 1989; Kapralski, 1993). Most combinatorics algorithms and programs, however, have employed classical, non-intelligent approaches. For advanced combinatorics problems, intelligent computing becomes necessary, and this is the major focus of this article.

Logic programming has been playing an important role in artificial intelligence. With much simplification, an abstraction of the human intelligence process is logic, and its computer realization is logic programming. Logic programming has been applied widely to every domain of intelligent computing, including knowledge-based systems, machine learning, data mining, scientific discovery, natural language processing, compiler writing, symbolic algebra, circuit analysis, relational databases, image processing, and molecular biology (Muggleton, 1999; Bratko, 2001). Logic programming is one of the best tools for working on intelligent computing in any form. This is why we integrate logic programming with combinatorics problems, which may be called *intelligent combinatorics*.

In the following, we discuss how the basic generating problems in combinatorics can be implemented in logic programming, especially in Prolog. We start with an overview on how combinatorics, particularly permutations and combinations, can be applied to specific problems in various fields. Computationally hard combinatorics problems are discussed to illustrate the usefulness of incorporating logic programming. They include the traveling salesman and job scheduling for permutations, and maximum independent set for combinations. Detailed discussions on the implementation of logic programming in Prolog follow. These programs can be readily integrated for intelligent computing approaches to solve combinatorial problems in many fields.

2. Application Areas of Combinatorics

Here we discuss past successful application domains that involve combinatorics with future potentials for incorporating logic programming. Since the field is extremely broad, we will consider only selected examples. Obviously, there are many other possibilities. This section serves as a brief survey of combinatorics applications in many fields.

2.1 ***Communication networks, cryptography and network security***. Permutations are frequently used in communication networks and parallel and distributed systems (Massini, 2003; Yang and Wang, 2004). Routing different permutations on a network for performance evaluation is a common problem in these fields. Many communication networks require secure transfer of information, which drives development in cryptography and network security (Kaufman, et al., 2003; Stallings, 2003). This area has recently become particularly significant because of the increased use of internet information transfers. Associated problems include protecting the privacy of transactions and other confidential data transfers and preserving the network security from attacks by viruses and hackers. Encryption process involves manipulations of sequences of codes such as digits, characters, and words. Hence, they are closely related to combinatorics, possibly with intelligent encryption process that can employ logic programming. For example, one common type of encryption process is interchanging--i.e., permuting parts of a sequence (Nandi, et al., 1994**)**. Permutations of fast Fourier transforms are employed in speech encryption (Borujeni, 2000).

2.2 ***Computer architecture***. Design of computer chips involves consideration of possible permutations of input to output pins. Field-programmable interconnection chips provide user programmable interconnection for a desired

permutation (Bhatia and Haralambides, 2000). Arrangement of logic gates is a basic element for computer architecture design (Tanenbaum, 1999).

2.3 ***Computational molecular biology***. This field involves many types of combinatorial and sequencing problems of items such as atoms, molecules, DNAs, genes, and proteins (*Combinatorial Pattern Matching,* 1992-2009; Doerge and Churchill, 1996; Chiang and Eisen, 2001; Siepel, 2003). One-dimensional sequencing problems are essentially permutation problems under certain constraints. One of the most successful domains of logic programming in terms of practicality is said to be molecular biology. Some of these practical applications include: formulating rules that accurately predict the activity of untried drugs; predicting the capacity of a chemical agent to cause permanent alteration of the genetic material within a living cell; and predicting the secondary structures of a protein given a sequence of amino acid residues (Muggleton, 1999).

2.4 ***Languages***. Both natural and computer languages are closely related to combinatorics (*Combinatorial Pattern Matching,* 1992-2009). This is because the components of these languages, such as sentences, paragraphs, programs, and blocks, are arrangements of smaller elements, such as words, characters, and atoms. For example, a string searching algorithm may rely on combinatorics of words and characters. Direct applications of this can include word processing and databases. Another important application area is performance analysis of these string searching algorithms. The study of computability--what we can compute and how it is accomplished--draws heavily on combinatorics. Logic programming has played important roles in natural and computer language processing, including parsing and compiler writing. The major reason for its prominence is because logic programming is a powerful tool for symbolic string and list processing. Logic programming is also useful for semantic analysis of languages. Hence, a combination of logic programming and combinatorics is a natural intersection, which can lead to many applications.

2.5 ***Pattern analysis***. In a broad sense, all the above-mentioned areas can be viewed as special cases of pattern analysis. Molecular biology, for example, studies patterns of atoms, molecules, and DNAs whereas languages treat patterns of sentences, words, and strings. Patterns can have many other forms; for example, visual images, acoustic signals, and other physical quantities such as electrical, pressure, temperature, etc. Patterns can also be abstract without any associated physical meaning. These patterns may be represented in various ways such as digital, analog, and other units. Some of these types of patterns can be associated with combinatorics. There has been extensive research on combinatorial pattern matching (*Combinatorial Pattern Matching,* 1992-2009). Computer music can be a specialized application domain of combinatorics of acoustic signals. Logic programming is a useful tool for pattern matching and analysis, including combinatorial ones.

2.6 ***Scientific discovery***. For certain types of knowledge discovery problems, generation of combinatorial sequences may become necessary in the process of yielding candidate solutions. For example, in scientific discovery, we may want to have a sequence of plausible chemical/biological reactions and their formations (Valdes-Perez, 1999). In each step of the sequence, we may generate combinatorial sequences of chemical/biological radicals, bases, and molecular compounds as candidate solutions and may select the most likely ones under certain rules and constraints. In another example, certain areas of mathematics, such as graph theory and number theory, may generate combinatorial sequences as candidate solutions.

2.7 ***Databases and data mining***. Queries in databases are multiple join operations that are permutations of the constituent join operations. Determining an optimal permutation that gives minimum cost is a common and important problem (Kumar Verma, and Trimbak Tamhankar, 1997). Data mining or knowledge discovery in databases is a relatively new field that aims at distilling useful information, often from large databases. In this area, techniques employing symbolic AI can manipulate combinatorial sequences of atoms or information elements. Non-symbolic knowledge discovery techniques, such as genetic algorithms and genetic programming, most commonly deal with solutions in the form of sequences of bits, digits, characters, and sometimes Lisp program elements . Neural networks, another domain of non-symbolic AI, sometimes deal with combinatorial patterns. Knowledge discovery techniques under uncertainty, such as Bayesian networks, Dempster-Shafer theory, fuzzy logic, and rough set theory, may have combinatorial solutions (Munakata, 1998a). Logic programming may fit well for intelligent manipulation of some of these combinatorial solutions.

2.8 ***Operations research***. Many optimization problems in operations research (OR) involve combinatorics. The job scheduling problem is essentially a sequencing problem to determine the order of jobs to be processed in an effort to

minimize the total time, cost, etc. Here, jobs can be in a computer system, network, or processing plant. Many problems involving graphs or networks also deal with the order of vertices and edges. The traveling salesperson problem is to determine the order of cities to be visited to minimize the total distance (Matsumoto and Yashiki, 1999). The shortest path problem of a graph is to determine a sequence of edges, the total length of which is minimum. Oftentimes, these problems are computationally difficult--e.g., NP-complete or NP-hard--and, therefore, require extensive research. It is quite conceivable that intelligent computing involving logic programming can make significant contributions in this field. For example, a hybrid system may integrate traditional OR and logic programming techniques. The latter can include knowledge-based and database systems, machine learning, natural language interface, symbolic algebra, network analysis, and pattern analysis. These techniques may help intelligent manipulation of the target data. For example, from a set of solution sequences, underlying rules may be derived and utilized for more efficient future computation. Deriving underlying rules from a set of patterns is quite common in logic programming (Bratko, 2001; Muggleton, 1999).

2.9 *Simulation*. Permutations and combinations can be employed for simulations in many areas. Permutations representing various genotype-phenotype associations are employed in genetics simulations (Doerge and Churchill, 1996). Other areas that employ permutations and combinations for simulations include networks, cryptography, databases and OR.

Other areas of applications include: complexity analysis, recursion, games, statistical mechanics, and electrical engineering (Liu, 1968).

3. Integrating Logic Programming with Application Problems of Permutations and Combinations

Most combinatorics algorithms and programs have been employing traditional approaches. For some advanced combinatorics problems, intelligent computing may become necessary. They include problems with complex constraints that cannot be easily implemented in the traditional approaches. In other cases solutions may require the use of background knowledge or inference processes. Logic programming is a typical approach to implement these intelligent computing techniques. In this section, we briefly discuss circumstances where intelligent computing may be required and how it can be implemented, particularly in terms of logic programming. While the concept can be applied to any combinatorics problems, we select well known, computationally hard examples.

3.1 Permutations

The traveling salesman problem (TSP) is an NP-complete, famous optimization problem for permutations (Garey, 1979). We are given n cities and a distance matrix $D = [d_{ij}]$, where d_{ij} is the distance between city i and city j. The problem is to determine the order of the cities to be visited, i.e., a permutation of 1 through n, expressed as $\pi(1)$, ..., $\pi(n)$, that minimizes the total distance of a tour, i.e., $\Sigma_{i=1}^{n-1} d_{\pi(i), \pi(i+1)} + d_{\pi(n), \pi(1)}$. The last term indicates that the tour must end at the originating city. The TSP and its variants have diverse practical applications such as vehicle routing, PCB design, and X-ray crystallography (Jung, 2002). The TSP has been chosen as a popular bench mark problem to test the effectiveness of many new techniques. The current techniques can be divided into two categories. One is exhaustive search and its improvements such as dynamic programming and branch-and-bound algorithms. Optimal solutions are often guaranteed for these techniques. The other category includes newer techniques such as the Hopfield-Tank neural network model and genetic algorithms (Munakata, 1998a). Typically optimal solutions are not guaranteed for the techniques in the second category. For either category, particularly for the first, generation of permutations may be necessary as a part of seeking solutions.

Extensions and variants can be in many forms with practical implications. There may be preferred sequences of cities in addition to minimizing the total distance. There may be different priorities on the cities to be visited. For example, an electric utility company may need efficient scheduling for vehicle routing for, say, 20 trucks for repair/maintenance work in a city. This type of problem is very common for daily execution in many industries such as transportation in the real world. The company needs to determine the most efficient routing of points within the city for each truck in the dynamically changing environment, day by day, or even minute by

minute. There may be preferred sequences of work points because, e.g., it may be more efficient to perform the same type of work consecutively. There may be different priorities on the points because their urgencies are different, e.g., emergency calls, minor repairs, and routine maintenance work. The entire process may require background knowledge in from of knowledge base as a collection of if-then rules. To extract new knowledge from the daily operations, inductive inference processes will be necessary.

Fig. 1 illustrates a simple example of the TSP, where letters are used for cities for easy identification. Without additional conditions, an optimal tour is *ABCEDA* with a total distance of 31. Suppose that *CD* or *DC* is a preferred sequence; then an optimal tour is *ABECDA* with a total distance of 35. If *CD* is the only preferred sequence, one can assign a temporary fake distance between *C* and *D*, say, 0.1 to solve the problem. The condition, however, can be much more complex, e.g., if *C* is followed by *D*, then *E* and *F* (for a problem with more cities) must not follow *D*, and so on. When the condition becomes complex, it would be impossible to solve the problem by simply manipulating the distance matrix. For such a problem, logic programming will be a useful tool. The condition can be expressed and imbedded in a Prolog program. For example, the condition: "if *C* is followed by *D*, then *E* and *F* must not follow *D*" can be expressed in form of "*C*, *D*, NOT (*E*, *F*)." To deal with more complex conditions or to employ background knowledge, a procedure call in form of p(<condition>, <action>) can be imbedded into the program. An "expert" procedure can advise a best <action> for a given <condition> based on its knowledge base. Prolog implementation of the TSP has also been discussed in the literature (Le, 1993; Bratko, 2001; WASP, 2005).

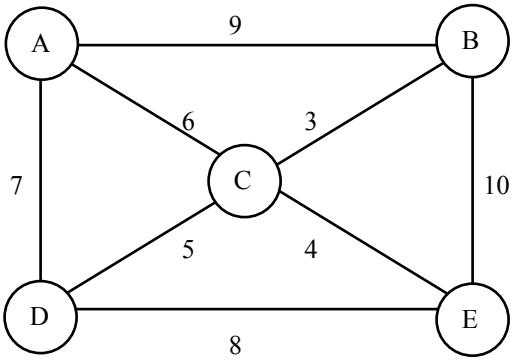

Fig. 1. Simple TSP example of five cities.

The same concept for the TSP can be applied to many other sequencing problems. The job scheduling problem is another NP-complete, famous optimization problem for permutations. We are given n jobs and corresponding processing time, p_i, $i = 1, n$, and m machines M_i, $i = 1, m$. The problem is to determine the order and assignments of the jobs to the machines so that the total processing time is minimized. Again, there are many variants of the problem, reflecting the popularity in the real world. The quantities involved can be either static or dynamic; or deterministic or probabilistic. Another variant is the tardiness problem, where the total penalty for tardiness is to be minimized. As an extension as in the case of the TSP, we can impose additional conditions such as, if job *C* is followed by job *D*, then jobs *E* and *F* must not follow job *D*, and so on. This is another example where logic programming may prove to be useful to solve the problem. In turn, these techniques can be applied to specific domains such as communication networks and computer architecture discussed earlier.

3.2 Combinations

The maximum independent set problem is an NP-complete, well known optimization problem for combinations. An independent set in a graph $G = (V, E)$, where V is the set of vertices and E is the set of edges, is a set of vertices where no two vertices are adjacent. A maximum independent set is an independent set whose cardinality is the largest among all independent sets of a graph. In Fig. 1, there are two maximum independent sets, {*A*, *E*} and {*B*,

D}. It turns out that two other popular problems, the maximum clique and vertex cover, are different versions of essentially the same problem (Garey, 1979). These problems have many real world applications including the following: finding ground states of spin glasses with exterior magnetic field and solving circuit layout design in VLSI circuits and printed circuit boards (Takefuji, 1992); information retrieval, experimental design, signal transmission, and computer vision (Balas, 1986); labeled pattern matching (Ogawa, 1986); PLA folding (Lecky, 1989); and stereo vision correspondence (Horaud, 1989).

As in the cases of the hard permutation problems discussed above, logic programming can be an effective tool for extensions of these combination problems. In a maximum independent set problem, additional constraints can be, if vertices *A* and *B* are included, vertices C and *D* must be included (even if they are adjacent), or if vertices *A* and *B* are included, vertices C and *D* must not be included (even if they are not adjacent), etc. These constraints may be easier to implement in logic programming than in ordinary algorithms.

4. Combinatorics Implementation in Logic Programming

The remainder of this article is primarily devoted to detailed discussions on implementing the four most basic problems in combinatorics using Prolog, the standard, and by far the most widely used, language for logic programming. These basic forms provide combinatorics tools in logic programming; that is, the bases of combining the two areas. Other forms may be derived by modifying the basic forms or developing similar programs. The programs discussed hereinafter generate all possible elements (for example, permutations). If only partial elements are required, they can be generated by placing the screening conditions within or outside the programs. Direct representation of combinatorics problems within Prolog programs as presented here, rather than Prolog programs calling external combinatorics generating functions written in another language such as C, will be advantageous in most cases for self-consistency and computational efficiency.

Generally, however, efficient solutions in Prolog are not necessarily trivial, and this is also the case for most of the problems in this article. In the following, Prolog implementations for the following four common combinatorics problems are presented: 1. Permutations; 2. Permutations with item repetitions; 3. Combinations; 4. Combinations with item repetitions. Previously, Prolog solutions for only a special case of Problem 1, permutations of n items taken from a pool of n (rather than more general r, where $r \leq n$) items, has been reported. For practical applications, these programs can readily be integrated into other Prolog programs.

We note that the order of the items in a *sequence* does matter, while it does not in a *set*. A permutation is a sequence of items, while a combination is a set of items. Generations of permutations or combinations may become necessary when each solution of an application problem is represented by a sequence or a set of items. Each solution of the application problem then will be a permutation or combination. The entire set of possible solutions will be a set of all possible permutations or combinations, which represents the upper-bound search space of the problem. A reader who is also interested in dealing with sets in Prolog may refer to (Munakata, 1992; Munakata, 1998).

All programs described here generate permutations or combinations in lexicographic order. For example, in lexicographic order, permutations of (1, 2) will be (1, 2), (2, 1), rather than (2, 1), (1, 2). Usually lexicographic is the most convenient way of organizing permutations or combinations. The term "complexity" refers to time complexity in the following.

5. Problem Descriptions

5.1 Notation.

Following the Prolog convention, variables in programs are denoted by upper-case letters, while variables in the main text follow the mathematics convention and are often represented by lower-case letters.

P: a pool of items (or elements) from which items are taken. A pool is represented by a

	Prolog list and items by list elements. For example, *P* can be [*a*, *b*, *c*], [ann, bob, cathy], or [1, 2, 3].
N:	the number of items in a list such as *P*. For example, for *P* = [*a*, *b*, *c*], we have *N* = 3.
R:	the number of items taken at a time.
M:	the number of ways (counts) taking *R* items at a time from a pool of *N* items.
X, *Y*:	items such as *a*, *b*, or 1, 2.
L, *L*1, etc.:	a list of items such as [*a*, *b*].
LL, *LL*1, etc:	a list of sublists, where each sublist contains items, e.g., [[*a*, *b*], [*c*, *d*]].

Given *P* or *N* and *R*, our problems are to determine *LL*.

5.2 Simple illustrations

For easy understanding of the four types of problems and their notations, we will briefly describe these problems then show simple examples (Liu, 1968). In the following *LL* is the list of sublists, where each sublist contains *R* items taken from a pool *P* of *N* items at a time. The cardinality, that is, the number of sublists in *LL* is *M*, defined above.

Let *P* = [*a*, *b*, *c*], *N* = 3, and *R* = 2.

1. Permutations

To *arrange R* items taken at a time from a pool of *N* items, where $R \leq N$. For our example,

M = 6, *LL* = [[*a*, *b*], [*a*, *c*], [*b*, *a*], [*b*, *c*], [*c*, *a*], [*c*, *b*]].

2. Permutations with item repetitions

Similar to ordinary permutations, described in Problem 1. The difference is that each item in the pool can be picked out any number of times. Because of this, *R* can be <, =, or > *N*.

M = 9, *LL* = [[*a*, *a*], [*a*, *b*], [*a*, *c*], [*b*, *a*], [*b*, *b*], [*b*, *c*], [*c*, *a*], [*c*, *b*], [*c*, *c*]].

3. Combinations

To *select R* items taken at a time from a pool of *N* items, where $R \leq N$. The difference between permutations and combinations is that the order of the items matters in the former while it does not in the latter. For our example,

M = 3, *LL* = [[*a*, *b*], [*a*, *c*], [*b*, *c*]].

4. Combinations with item repetitions

Similar to ordinary combinations, described in Problem 3. The difference is that each item in the pool can be picked out any number of times. Because of this, *R* can be <, =, or > *N*.

M = 6, *LL* = [[*a*, *a*], [*a*, *b*], [*a*, *c*], [*b*, *b*], [*b*, *c*], [*c*, *c*]].

Mathematical formulas for *M* are given as follows ("*c*" in *cnPr*, for example, stands for "count") (Liu, 1968). Since Prolog implementations of these formulas are straightforward, they are not presented here.

Permutations: c$nPr = n!/(n - r)! = n(n - 1)\ldots(n - r + 1)$.
Permutations with item repetitions: c$n\Pi r = n^r$.
Combinations: cnCr = c$nPr/r! = n!/(n - r)!r! = n(n - 1)\ldots(n - r + 1)/r(r - 1)\ldots1$.
If $r > n - r$, use c$nCn\text{-}r$ = c$nPn\text{-}r/(n - r)! = n(n - 1)\ldots(r + 1)/(n - r)(n - r - 1)\ldots1$.
Combinations with item repetitions: cnHr = c$(n+r\text{-}1)Cr$.

6. Combinatorics Generations

6.1 Preliminaries

6.1.1 Representation of items (elements)

Generally, items can be represented in various ways such as [adams, brown, carter], or simply [*a*, *b*, *c*] or [1, 2, 3]. The programs in this article work for any form of item representation. We use the letter representation of [*a*, *b*, ...] for illustration.

6.1.2 Utility procedures

The following two basic procedures will be used. Variables in programs are not italicized.

```
% deletex(L, X, L1) deletes element X from L giving L1. e.g., deletex([a, b], b, [a]).

deletex([X | Lt], X, Lt).
deletex([X | Lt], Y, [X | Ls]) :- deletex(Lt, Y, Ls).

% addx(LL, X, LLa, LL1) first inserts element X at the beginning of every element list
% of LL then this resulting list is appended by LLa giving LL1.
% e.g., addx([[a, b], [c, d]], x, [[e, f], [g, h]], [[x, a, b], [x, c, d], [e, f], [g, h]]).

addx([ ], _, LLa, LLa).
addx([L | LLt], X, LLa, [[X | L] | LL1]) :- addx(LLt, X, LLa, LL1).
```

The second clause of deletex is recursively called until *X* is found. On the average it requires | *L* | / 2 search. Hence the complexity of deletex is O(| *L* |). In addx, starting from *LL* the second clause is recursively called until it becomes [ ]. The complexity of addx is O(| *LL* |).

In the remainder of this article, although standard definitions of *nPr*, *nCr*, and so on, are the *number* of permutations, combinations, and so on, we use these expressions as "icons" to represent *permutations and combinations themselves* (e.g., [[1, 2], [2, 1]]).

6.2 *nPr*: Permutations, *R* items out of *N* items

The following program generates list *LL* of sublists, where each sublist is a permutation of *R* items taken at a time from a pool *L* of *N* items. We recall $R \le N$. The generated permutations in *LL* are in lexicographic order with respect to the original order of the items in *L* (for example, if *L* = [*a*, *b*, *c*], then "*a*, *b*, *c*" is the original order). Generally, the order of permutations does not matter (for example, [[*a*, *b*], [*b*, *a*]] is the same as [[*b*, *a*], [*a*, *b*]]). However, placing them in lexicographic order (as for example, [[*a*, *b*], [*b*, *a*]]) is easy to track and most common in practice, and we follow this convention throughout this article. The *nPr* program requires procedures deletex and addx defined in Section 4. A special case of *nPr* where *N* = *R*, i.e., *nPn* is a common combinatorics problem whose solutions are found in Prolog books (Bratko, 2001; Le, 1993).

The basic idea of *nPr*(*L*, *R*, *LL*) is to divide *LL* into two groups, Groups 1 and 2, determine each group, and append them to yield *LL*. Group 1 contains permutations starting with *X*, the head of *L*. Group 2 contains permutations not starting with *X*. In permsub, Group 1 is obtained by first separating *X* from *L* (by the deletex call), taking permutations of the remaining elements of *L* with length *R* – 1 (by the *nPr* call), then putting back *X* at the beginning of every permutation obtained by the *nPr* call (as a part of the addx call). Group 2 is obtained by the permsub call. In general permsub(*Ls*, *L*, *R*, *LL*), where *Ls* is a subset of *L*, generates all permutations of *R* elements starting with an element in *Ls* followed by all permutations of length *R* – 1 consisting of the remaining elements in *L*, giving *LL*. For Group 2, we call permsub(*Lt*, *L*, *R*, *LL2*) where *Lt* is the tail of *L* and *LL2* represents Group 2.

For example, for *nPr*([*a*, *b*, *c*], 2, *LL*), Group 1 is obtained by first separating *X* = *a* from *L* = [*a*, *b*, *c*] resulting [*b*, *c*], taking permutations of [*b*, *c*] with length *R* – 1 = 1 yielding [[*b*], [*c*]], then putting back a at the beginning of every permutation in [[*b*], [*c*]], giving [[*a*, *b*], [*a*, *c*]]. Group 2 is obtained by permsub([*b*, *c*], [*a*, *b*, *c*], 2, *LL*2), which yields *LL*2 = [[*b*, *a*], [*b*, *c*], [*c*, *a*], [*c*, *b*]]. Finally, the two groups are appended yielding [[*a*, *b*], [*a*, *c*], [[*b*, *a*], [*b*, *c*], [*c*, *a*], [*c*, *b*]].

```
% nPr(L, R, LL) generates permutations of elements of L, taken R elements
% at a time giving LL. e.g., nPr([a, b], 2, [[a, b], [b, a]]).

nPr(_, 0, [[ ]]).
nPr(L, R, LL) :-
        R >= 1,
        permsub(L, L, R, LL).

% permsub(Ls, L, R, LL), where Ls is a subset of L, generates all permutations of
% R elements starting with an element in Ls followed by all permutations of
% length R – 1 consisting of the remaining elements in L, giving LL.
% e.g., permsub([b, c], [a, b, c], 2, [[b, a], [b, c], [c, a], [c, b]]).

permsub([ ], _, _, [ ]).            % When no element in Ls, no permutation in LL.
permsub([X | Lt], L, R, LL) :-
        R1 is R - 1,
        deletex(L, X, L1),          % Separates head X of Ls from L and
        nPr(L1, R1, LL1),           % gets permutations of the remaining elements.
        permsub(Lt, L, R, LL2),     % LL2 contains permutations not starting with X.
        addx(LL1, X, LL2, LL).      % Inserts X into the LL1 permutations and append LL2.
```

6.2.1 Complexity analysis nPr(L, R, LL)

We will determine the time complexity of procedure *nPr*(*L*, *R*, *LL*) as *f*(*n*, *R*), where *n* = | *L* |. When *nPr*(*L*, *R*, *LL*) is called, it invokes the second clause of permsub (except a trivial case of *L* = [ ] for which the first clause, i.e., the boundary condition, of permsub is invoked). Within the second clause, four procedures, deletex, *nPr*, permsub and addx, are called. The complexity of deletex is O(*n*) as discussed before, and is negligible in comparisons with the others. The complexity of addx O(| *LL*1 |) is, as we will see soon, at most the complexity of the *nPr* call and it can be included as a part of *nPr*. This leaves only two recursive calls, *nPr* and permsub within the second clause of permsub.

Let us use notation of *nPr*{*n*, *R*, _} and permsub{*n*, *n*, *R*, _} to represent the list sizes or magnitudes of the arguments. For example, *n* and *R* in *nPr*{*n*, *R*, _} represent *n* = | *L* | and *R* = *R* in an *nPr*(*L*, *R*, *LL*) call. When *nPr*{n, *R*, _} is invoked at the beginning, it calls the second clause of permsub{*n*, *n*, *R*, _}. In turn, *nPr*{*n* – 1, *R* – 1, _} and permsub{*n* – 1, *n*, *R*, _} are recursively called. When we draw a search tree for *nPr*{*n*, *R*, _} and focus only on permsub for the moment, the branch extends as permsub{*n*, *n*, *R*, _}, permsub{*n* - 1, *n*, *R*, _}, …, permsub{0, *n*, *R*, _}. The number of nodes so far is *n* + 2, which consists of *n* – 1 permsub nodes plus one *nPr*{*n*, *R*, _} at the root.

We note that each of these permsub calls, except the last call permsub{0, *n*, *R*, _}, invokes *nPr*{*n* – 1, *R* – 1, _}, that is, there are a total of *n nPr*{*n* – 1, *R* – 1, _} invocations in the search tree. This leads to the following recurrence equation for *f*(*n*, *R*) as the number of nodes in the search tree.

$$f(n, R) = n * f(n - 1, R - 1) + (n + 2), f(\_, 0) = 1.$$

The boundary condition corresponds to the first clause of *nPr*. The last two terms of the right hand side, *n* + 2, contribute at most the same as the first term to determine the order of the complexity. Hence, it is sufficient to consider the following homogeneous version of the recurrence equation for our purpose.

$$f(n, R) = n * f(n - 1, R - 1), f(\_, 0) = 1.$$

This equation can be solved as

$$f(n, R) = n * f(n - 1, R - 1) = n(n - 1) * f(n - 2, R - 2) = \ldots = n(n - 1)\ldots(n - R + 1) * f(n - R, 0)$$
$$= n! / (n - R)! = {}_nP_R.$$

That is, the complexity of procedure *nPr*(*L*, *R*, *LL*) is O(*n*! / (*n* – *R*)!) = O($_nP_R$). The last expression $_nP_R$ is the number of permutations, *R* items taken at a time from *n* items, and not the Prolog procedure *nPr*. This is a reasonable consequence since permutations are generated one by one by the program and there are $_nP_R$ permutations all together. We also note that the complexity of addx O(| *LL*1 |) is at most the complexity of the *nPr* recursive call and the addx call can be included as a part of the *nPr* call.

6.3 *nPIr*: Permutations with item repetitions, *R* items out of *N* items

The following program generates list *LL* of sublists, where each sublist is a permutation of *R* items taken at a time from a pool *L* of *N* items. Each of the *N* items can be repeated any number of times in a permutation. We recall *R* can be <, =, or > *N*. The generated permutations in *LL* are in lexicographic order. The program requires procedures deletex and addx defined earlier.

The basic idea of *nPIr*(*L*, *R*, *LL*) is similar to *nPr*(*L*, *R*, *LL*). *LL* is divided into two groups, Groups 1 and 2, and they are appended to yield *LL*. Group 1 contains permutations starting with *X*, the head of *L*. Group 2 represents permutations not starting with *X*. For example, for *nPIr*([*a*, *b*, *c*], 2, *LL*), Group 1 is obtained by taking permutations of [*a*, *b*, *c*] with length *R* – 1 = 1 yielding [[*a*], [*b*], [*c*]], then putting back *a* at the beginning of every permutation, giving [[*a*, *a*], [*a*, *b*], [*a*, *c*]]. The major difference of *nPIr* from *nPr* is that we take permutations of length *R* – 1 of *L* = [*a*, *b*, *c*] rather than *Lt* = [*b* ,*c*], thus the element *X* = *a* is repeated. Group 2 represents permutations not starting with *X* = *a*, and it is obtained by pisub([*b*, *c*], [*a*, *b*, *c*], 2, *LL*2), which yields *LL*2 = [[*b*, *a*], [*b*, *b*], [*b*, *c*], [*c*, *a*], [*c*, *b*], [*c*, *c*]]. Finally, the two groups are appended yielding [[*a*, *a*], [*a*, *b*], [*a*, *c*], [*b*, *a*], [*b*, *b*], [*b*, *c*], [*c*, *a*], [*c*, *b*], [*c*, *c*]].

```
% nPIr(L, R, LL) generates permutations of L with element repetition, taken R
% elements at a time giving LL. e.g., nPIr([a, b], 2, [[a, a], [a, b], [b, a], [b, b]]).

nPIr(_, 0, [[ ]]).
nPIr(L, R, LL) :-
        R >= 1,
        pisub(L, L, R, LL).

% pisub(Ls, L, R, LL), where Ls is a subset of L, generates all sequences of R elements
% starting with an element in Ls followed by length R – 1, element-repeated permutations
% of L, giving LL.
% e.g., pisub([b, c], [a, b, c], 2, [[b, a], [b, b], [b, c], [c, a], [c, b], [c, c]]).
```

```
pisub([ ], _, _, [ ]).
pisub([X | Lt], L, R, LL) :-
        R1 is R - 1,
        nPIr(L, R1, LL1),          % Generates element-repeated permutations of length R – 1.
        pisub(Lt, L, R, LL2),      % LL2 has permutations of length R not starting with X.
        addx(LL1, X, LL2, LL).     % Inserts X at the beginning of the LL1 permutations, appends
                                       LL2.
```

6.3.1 Complexity analysis nPIr(L, R, LL)

This is similar to the analysis of *nPr*(*L*, *R*, *LL*) discussed earlier. The only differences of *nPIr*(*L*, *R*, *LL*) from *nPr*(*L*, *R*, *LL*) are in the second clause of pisub, there is no deletex and there is a recursive call *nPIr*{*n*, *R* – 1, _} instead of *nPr*{*n* – 1, *R* – 1, _}. This leads to the following recurrence equation for *f*(*n*, *R*) as the number of nodes in the search tree.

$$f(n, R) = n * f(n, R – 1) + (n + 2), f(\_, 0) = 1.$$

As before, it is sufficient to consider the following homogeneous version of the recurrence equation for our purpose.

$$f(n, R) = n * f(n, R – 1), f(\_, 0) = 1.$$

This equation can be solved as

$$f(n, R) = n * f(n, R – 1) = n^2 * f(n, R – 2) = \ldots = n^R * f(n, 0) = n^R = {}_n\Pi_R.$$

That is, the complexity of procedure *nPIr*(*L*, *R*, *LL*) is $O(n^R) = O({}_n\Pi_R)$. Again, this is a reasonable consequence since each permutation is generated by the program and there are ${}_n\Pi_R$ permutations all together. For this problem, the inhomogeneous version of the original recurrence equation can also be solved as follows.

$$f(n, R) = n * f(n, R – 1) + (n + 2) = n^2 * f(n, R – 2) + n(n + 2) + (n + 2) = \ldots$$
$$= n^R * f(n, 0) + (n^{R-1} + \ldots + n + 1)(n + 2)$$

$$= \begin{cases} n^R + (n + 2)(n^R – 1) / (n – 1) & (n \neq 1) \\ 1 + 3R & (n = 1) \end{cases}$$

6.4 *nCr*: Combinations, *R* items out of *N* items

The following program generates list *LL* of sublists, where each sublist is a combination of *R* items taken at a time from a pool *L* of *N* items. We recall $R \leq N$. The generated combinations in *LL* are in lexicographic order, in terms of both order of combinations as well as order of items within each combination. We note that generally these orders do not matter; for example, for the former, [[*a*, *b*], [*a*, *c*]] = [[*a*, *c*], [*a*, *b*]] and for the latter, [*a*, *b*] = [*b*, *a*]. However, lexicographic order is easy to track and most common in practice and we follow this convention. Our coding scheme is based on lexicographic order. The *nCr* program requires procedure addx defined earlier.

The basic idea of *nCr*(*L*, *R*, *LL*) is similar to *nPr*(*L*, *R*, *LL*) and *nPIr*(*L*, *R*, *LL*). *LL* is divided into two groups, Groups 1 and 2, and they are appended to yield *LL*. Group 1 contains combinations starting with *X*, the head of *L* (assuming the combinations are generated in lexicographic order). Group 2 represents combinations not starting with *X* (i.e., not containing *X*). For example, for *nCr*([*a*, *b*, *c*], 2, *LL*), Group 1 is obtained by taking combinations of [*b*, *c*] with length *R* – 1 = 1 yielding [[*b*], [*c*]], then putting back a at the beginning of every combination, giving [[*a*, *b*], [*a*, *c*]]. Group 2 is combinations not containing *X* = *a*, and it is obtained by

combinations of [*b*, *c*] of length *R*, i.e., [[*b*, *c*]]. Finally, the two groups are appended yielding [[*a*, *b*], [*a*, *c*], [*b*, *c*]].

```
% nCr(L, R, LL) generates combinations of elements of L, taken R elements
% at a time giving LL. e.g., nCr([a, b, c], 2, [[a, b], [a, c], [b, c]]).

nCr(L, R, LL) :-
        length(L, N),
        N >= R,
        combsub(L, N, R, LL).

% combsub(L, N, R, LL) generates combinations of elements of list L of size N,
% taken R elements at a time giving LL.

combsub(_, _, 0,[[ ]]) :- !.          % When no element is taken at a time, an empty combination in
LL.
combsub(L, N, N,[L]) :- !.            % When R = N, there is only one combination, i.e., L itself.
combsub([X | Lt], N, R, LL) :-
            R1 is R - 1,
            N1 is N - 1,
            combsub(Lt, N1, R1, LL1),     % Generates combinations of length R – 1 without X.
            combsub(Lt, N1, R, LL2),      % LL2 has combinations of length R not having X.
            addx(LL1, X, LL2, LL).        % Inserts X at the beginning of the LL1 combinations,
                                              appends LL2.
```

In the above *nCr* calls combsub once and all the remaining computation is performed within combsub. One might wonder whether the two procedures *nCr* and combsub can be merged into one procedure but this is not the case. To make *nCr* as simple as possible, we want to keep the number of arguments minimum, which is three for *L*, *R* and *LL*. However, we need one more argument $N = | L |$, which is determined internally in *nCr* and used in combsub. We need *N* since when $R = N$, the answer should be simply [*L*] as a boundary condition (the second clause of combsub).

6.4.1 Complexity analysis nCr(L, R, LL)

An application of similar previous analyses for the complexity $f(n, R)$ to $nCr(L, R, LL)$ leads to the following recurrence equation.

$$f(n, R) = f(n - 1, R) + f(n - 1, R - 1) + 1,\ f(n, n) = f(\_, 0) = 1.$$

The homogeneous version without the last term 1 is,

$$f(n, R) = f(n - 1, R) + f(n - 1, R - 1),\ f(n, n) = f(\_, 0) = 1.$$

This is a well known recurrence equation for the binomial coefficients or the Pascal's triangle, and its solution is

$$f(n, R) = n!/(n - R)!R! = {}_nC_R.$$

That is, the complexity of procedure $nCr(L, R, LL)$ is $O(n!/(n - R)!R!) = O({}_nC_R)$. Again, this is a reasonable consequence since there are ${}_nC_R$ combinations all together. For this problem, the inhomogeneous version of the original recurrence equation can also be found straightforward as $f(n, R) = 2n!/(n - R)!R! - 1 = 2{}_nC_R - 1$.

6.5 *nHr*: Combinations with item repetitions, *R* items out of *N* items

The following program generates list *LL* of sublists, where each sublist is a combination of *R* items taken at a time from a pool *L* of *N* items. Each of the *N* items can be repeated any number of times in a combination. We recall *R* can be <, =, or > *N*. The generated combinations in *LL* are in lexicographic order. Requires procedure addx defined earlier.

Again the basic idea of *nHr*(*L*, *R*, *LL*) is similar to the previous programs, particularly *nCr*(*L*, *R*, *LL*). *LL* is divided into two groups, Groups 1 and 2, and they are appended to yield *LL*. Group 1 contains combinations starting with *X*, the head of *L* (assuming the combinations are generated in lexicographic order). Group 2 contains combinations not starting with *X* (i.e., not containing *X*). For example, for *nHr*([*a*, *b*, *c*], 2, *LL*), Group 1 is obtained by taking combinations of [*a*, *b*, *c*] with length *R* – 1 = 1 yielding [[*a*], [*b*], [*c*]], then putting back *a* at the beginning of every combination, giving [[*a*, *a*], [*a*, *b*], [*a*, *c*]]. The major difference of *nHr* from *nCr* is that we take combinations of length *R* – 1 of *L* = [*a*, *b*, *c*] rather than *Lt* = [*b* ,*c*], thus the element *X* = *a* is repeated. Group 2 represents combinations not containing *X* = *a*, and it is obtained by combinations of [*b*, *c*] of length *R*, i.e., [[*b*, *b*], [*b*, *c*], [*c*, *c*]]. Finally, the two groups are appended yielding [[*a*, *a*], [*a*, *b*], [*a*, *c*], [*b*, *b*], [*b*, *c*], [*c*, *c*]]. Unlike *nCr*, *nHr* does not require additional procedures (except addx) since it does not need internally determined arguments such as *N*.

```
% nHr(L, R, LL) generates combinations of elements of L with repetition,
% taken R elements at a time giving LL. e.g.,
% nHr([a, b, c], 2, [[a, a], [a, b], [a, c], [b, b], [b, c], [c, c]]).

nHr([ ], _,[ ]) :- !. % When there is no element in the pool, no combinations in LL.
nHr(_, 0,[[ ]]) :- !.          % When no element is taken at a time, an empty combination in LL.
nHr([X | Lt], R, LL) :-
        R1 is R - 1,
        nHr([X | Lt], R1, LL1),  % Generates element-repeated combinations of length R – 1.
        nHr(Lt, R, LL2),         % LL2 has combinations of length R not having X.
        addx(LL1, X, LL2, LL). % Inserts X at the beginning of the LL1 combinations, appends
                                   LL2.
```

6.5.1 Complexity analysis nHr(L, R, LL)

Again, an application of similar previous analyses for the complexity *f*(*n*, *R*) to *nHr*(*L*, *R*, *LL*) leads to the following recurrence equation.

$$f(n, R) = f(n, R - 1) + f(n - 1, R) + 1, \text{ f}(0, \_) = f(\_, 0) = 1.$$

The homogeneous version without the last term 1 is,

$$f(n, R) = f(n, R - 1) + f(n - 1, R), f(0, \_) = f(\_, 0) = 1.$$

This is a slightly modified version of the well known recurrence equation for the binomial coefficients or the Pascal's triangle, and its solution is

$$f(n, R) = (n + R)! / (n!\, R!) = {}_{n+R}C_R = {}_{n+1}H_R.$$

That is, the complexity of procedure *nHr*(*L*, *R*, *LL*) is O((*n* +*R*)! / (*n*! *R*!)). We note that ${}_{n+1}H_R \ / \ {}_nH_R = (n + R) / n = 1 + R / n$, which is close to O(1) for many values of (*n*, *R*). Hence, this is a reasonable consequence considering there are ${}_nH_R$ combinations all together. For this problem, the solution for the inhomogeneous version of the original

recurrence equation can also be found as $f(n, R) = 2(n+R)! / (n! R!) - 1 = 2_{n+R}C_R - 1$.

7. Conclusion

The complexities of the programs for the four basic types of permutations and combinations presented are the same or close to their basic mathematical requirements (for example, to generate all permutations of R items taken from a pool of n items, it requires $n! / (n - R)!$ computations). Hence, the programs should be optimal or near optimal in terms of the order of their complexities. These programs can readily be employed for intelligent approaches for advanced combinatorics problems, involving processes such as inference and the use of background knowledge.

References

Akl, Selim G. (1989). *The Design and Analysis of Parallel Algorithms*, Englewood Cliffs, New Jersey, Prentice-Hall, Chapter 6.

Balas, Egon & Yu, Chung Sung. (1986). Finding a maximum clique in an arbitrary graph, *SIAM J. Comput*., 15(4), 1054-1068.

Bhatia Dinesh & Haralambides, James. (2000). Resource requirements and layouts for field programmable interconnection chips, *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 8(3): 346-55.

Borujeni. Shahram, Etemadi. (2000). Speech encryption based on fast Fourier transform permutation, *Proceedings of the 7th IEEE International Conference on Electronics, Circuits and Systems* (ICECS 2000), Dec. 17-20, 290-293, Jounieh, Lebanon, IEEE.

Bratko, Ivan. (2001). *Prolog Programming for Artificial Intelligence*, 3rd, Ed., Wokingham, England, Addison-Wesley.

Chiang, Derek Y. Brown, Patrick, O. & Eisen, Michael B. (2001). Visualizing associations between genome sequences and gene expression data using genome-mean expression profiles, *Bioinformatics*, 17(S1): 49-55.

*Combinatorial Pattern Matching;*In: *Proceedings of Annual Symposiums. Lecture Notes in Computer Science.* (1992-2000). Springer, Berlin.

Doerge R.W. & G. A. Churchill. (1996). Permutation tests for multiple loci affecting a quantitative character, *Genetics*, 142: 285-294.

Garey Michael R. & Johnson, David S. (1979). *Computers and Intractability : A Guide to the Theory of NP-Completeness*, San Francisco, W. H. Freeman.

Horaud Radu. & Skordas, Thomas.. (1989). Stereo correspondence through feature grouping and maximal cliques, *IEEE Transactions on Pattern Analysis*, *Machine Intelligence*, 11(11), 1168-1180.

Jung, Soonchul & Moon, Byung-Ro. (2002). Toward minimal restriction of genetic encoding and crossovers for the two-dimensional Euclidean TSP, IEEE *Transactions on Evolutionary Computation,* 6(6), 557-565.

Kapralski, Adam. (1993). New methods for generation of permutations, combinations, and other combinatorial objects in parallel, *Journal of Parallel and Distributed Computing* 17(4): 315-326.

Kaufman, Charlie, Perlman, Radia & Speciner, Mike. (2003). *Network Security*: *Private Communication in a Public World*, 2nd Ed., Upper Saddle River, NJ, Prentice Hall.

Kumar Verma, Ajit & Trimbak Tamhankar, Mangesh.. (1997). Reliability-based optimal task-allocation in distributed-database management systems, *IEEE Transactions on Reliability*, 46(4): 452-459.

Le, Tu Van. (1993). *Techniques of Prolog Programming with Implementation of Logical Negation and Quantified Goals*, 38, New York, Wiley.

Lecky John E., Murphy, Owen J., & Absher Richard G., (1989), Graph theoretic algorithms for the PLA folding problem, *IEEE Transactions on Computer-Aided Design*, 8(9), 1014-1021.

Liu, Chung Laung. (1968). *Introduction to Combinatorial Mathematics*, Computer Science Series, New York, McGraw-Hill, Chapter 1.

Massini. Annalisa. (2003). All-to-all personalized communication on multistage interconnection networks, *Discrete*

Applied Mathematics, 128(2-3): 435-46.
Matsumoto Naoki & Yashiki, Satoshi. (1999). Simple approach to TSP by permutation of six cities and deletion of crossover, *IEEE Pacific Rim Conference on Communications, Computers and Signal Processing*, Aug. 22-24, 377-380, Victoria, BC, Canada, IEEE.
Muggleton, Stephen. (1999). Scientific knowledge discovery using inductive logic programming, *Communications of the ACM*, 42(11): 42-46.
Munakata. Toshinori. (1992). Notes on implementing sets in Prolog, *Communications of the ACM*, 35(3): 112-120.
Munakata. Toshinori. (1998). Notes on implementing fuzzy sets in Prolog, *Fuzzy Sets and Systems*, 98(3): 311-317.
Munakata. Toshinori. (1998a). *Fundamentals of the New Artificial Intelligence: Beyond Traditional Paradigms*, New York, Springer-Verlag.
Mycielski, Jan, et al. (Eds.), (1997). *Structure in Logic and Computer Science*, Berlin, Springer-Verlag.
Nandi, S., B.K. Kar, & P. Pal Chaudhuri. (1994). Theory and applications of cellular automata in cryptography, *IEEE Transactions on Computers*, 43(12): 1346-1357.
Ogawa H. (1986). Labeled point pattern matching by Delaunay triangulation and maximal cliques, *Pattern Recognition*, 19(1), 35-40.
Roberts, Fred S. (1984). *Applied Combinatorics*, Englewood Cliffs, New Jersey, Prentice-Hall.
Siepel, Adam C. (2003). An algorithm to enumerate sorting reversals for signed permutations, *Journal of Computational Biology*, 10(3-4): 575-597.
Stallings, William. (2003). *Cryptography and Network Security*: *Principles and Practice*, Upper Saddle River, NJ, Prentice Hall.
Takefuji, Yoshiyasu. (1992). *Neural Network Parallel Computing*, Boston, MA, Kluwer Academic,
Tanenbaum, Andrew S. (1999). *Structured Computer Organization*, 4th Ed., Upper Saddle River, NJ, Prentice Hall.
Valdes-Perez, Raul E. (1999). Discovery techniques for scientific apps, *Communications of the ACM*, 42(11): 37-41.
WASP (Working group on Answer Set Programming). (2005). WASP-Showcase: Knowledge-based planning, http://www.kr.tuwien.ac.at/projects/WASP/planning.html
Yang, Yuanyuan. & Wang, Jianchao. (2004). Routing permutations on optical baseline networks with node-disjoint paths, *Proceedings of Tenth International Conference on Parallel and Distributed Systems* (ICPADS 2004), July 7-9, N.- Tzeng, F. (Ed), 65-72, Newport Beach, CA, IEEE.