COMBINATORICS IN LOGIC PROGRAMMING: IMPLEMENTATIONS AND APPLICATIONS

Toshinori MUNAKATA[1], **Roman BARTÁK**[2]

[1]*Computer and Information Science Department,*
Cleveland State University, Cleveland, Ohio 44115, USA
E-mail: t.munakata@csuohio.edu

[2]*Department of Theoretical Computer Science and Mathematical Logic,*
Charles University, Praha, Czech Republic
E-mail: bartak@kti.mff.cuni.cz

Abstract

This paper presents a new intelligent computing approach for combinatorics problems by incorporating logic programming. Permutations, one of the most common and basic topics in combinatorics, appear in many problems in science, engineering, and business. Applications of permutations and other combinatorics problems are briefly reviewed. Implementation of permutations is presented in Prolog, the standard language of logic programming. Time complexity analysis and experimental results of running the program are also discussed. The program is optimal in terms of the order of its complexity. Applications of the technique to various domains as well as to specific problems such as the traveling salesman are discussed.

Keywords: intelligent combinatorics problems, logic programming implementation

1 Introduction

Combinatorics, or combinatorial theory, is a major mathematics branch that has extensive applications in many fields. They include engineering, computer science, natural and social sciences, biomedicine, operations research,

and business [5]. Particular areas that have extensive applications of combinatorics such as permutations and combinations include: communication networks, cryptography and network security; computer architecture; electrical engineering; computational molecular biology; languages both natural and computer; pattern analysis; scientific discovery; databases and data mining; scheduling problems in operations research; and simulation. Other areas of applications include: complexity analysis, recursion, games, and statistical mechanics.

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 has been studied extensively because of the fundamental nature and the importance in practical applications. 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 paper.

Logic programming has been playing an important role in intelligent computing. 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. It is one of the best tools to work on any form of intelligent computing, and this is why we integrate logic programming with combinatorics problems. In the following, we discuss how the basic generating problems in combinatorics can be implemented in logic programming, especially in Prolog. Real world hard combinatorics problems are discussed to illustrate the usefulness of the logic programming approach.

2 Combinatorics Implementation in Logic Programming

In the following, Prolog implementation for permutations is presented that generates all possible elements (permutations). If only partial elements are required, they can be generated by placing the screening conditions within or outside the programs. Previously, Prolog solutions for only a special case

of permutations of n items taken from a pool of n (rather than more general r, where $r \leq n$) items, has been reported. Other common combinatorics problems can be implemented and analyzed in similar ways. These problems include: permutations with item repetitions; combinations; and combinations with item repetitions. For practical applications, these programs can readily be integrated into other Prolog programs. A reader who is also interested in dealing with sets in Prolog may refer to [6], [7].

The program described here generates permutations 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.

2.1 Preliminaries

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.

Utility procedures

The following two basic procedures will be used.

```
% 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).
```

In the following, although standard definitions of *nPr* is the *number* of permutations, we use this expression as an "icon" to represent *permutations themselves* (e.g., [[1, 2], [2, 1]]).

2.2 *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 $R \leq N$. A special case of *nPr*, where $N = R$, i.e., *nPn* is a common combinatorics problem whose solutions are found in Prolog books [1], [4]. In the next section we will show that the complexity of procedure *nPr*(*L*, *R*, *LL*) is $O(n!\ /\ (n - R)!) = O({}_nP_R)$. Hence, the order of the complexity is optimal.

```
% 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([ ], _, _, [ ]).
permsub([X | Lt], L, R, LL) :- R1 is R - 1, deletex(L, X, L1),
 permsub(Lt, L, R, LL2), addx(LL1, X, LL2, LL).
```

3 Complexity Analysis

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 $O(|LL1|)$ 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, $n, R, \_\}, \ldots,$ 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.

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

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.

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

This equation can be solved as

$$\begin{aligned} f(n, R) &= n * f(n - 1, R - 1) = ... = n(n - 1)...(n - R + 1) * f(n - R, 0) \\ &= n!/(n - R)! =_n P_R. \end{aligned} \quad (3)$$

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(| $LL1$ |) is at most the complexity of the *nPr* recursive call and the addx call can be included as a part of the *nPr* call.

4 Experimental Experience

Figure 1 shows the experimental test result of running time to compute *nPr* by employing two approaches called "classic" and "findall". The former is the program discussed in Section 2 which generates all permutations at

once. The latter is a straightforward permutation generator based on the Prolog built-in procedure findall. Findall is used to collect all permutations by repeated calling of the code producing one permutation at time (and the next permutation upon backtracking). For each value of $n = 1$ to 9, nPr is computed for $r = 1$ to n. For example, ${}_9P_r$ for $r = 1$ to 9 took about 1 second for the classic program. The graph in Figure 1 clearly demonstrates that the classic is more efficient than the findall approach and that the efficiency gap expands as n increases. Hence, it is useful to have a dedicated code for generating all permutations rather than re-using the code producing a single permutation at time. The experiment was performed on a 1.7 GHz PC with a Mobile Intel Pentium 4-M processor and 768 MB RAM running Windows XP Professional. The SICStus Prolog 3.8.7 compiler was employed. The experimental time for the classic is close to that expected from our complexity analysis in the previous section.

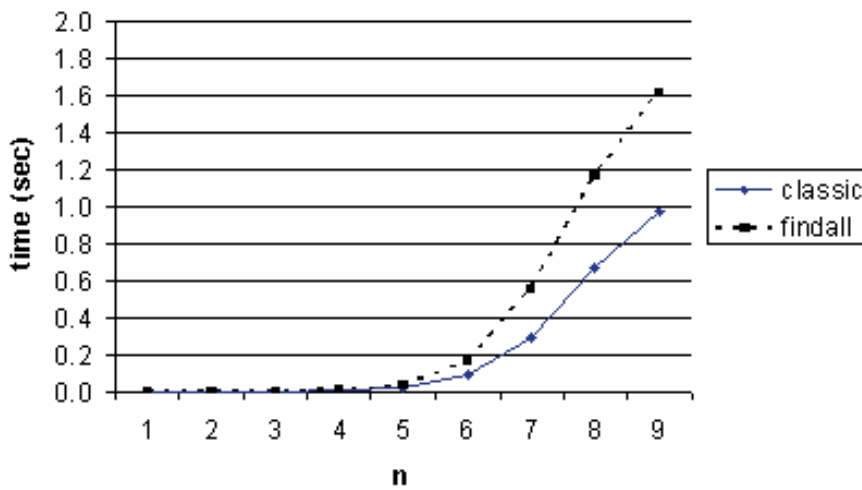

Figure 1. Experimental computing time for nPr. Time in seconds represents runtime for nPr, $r = 1$ to n, for each value of $n = 1$ to 9, employing two programs, the classic given in Section 2, and the standard procedure findall

5 Integrating Logic Programming with Application Problems of Permutations and Combinations

Most combinatorics algorithms and programs have been employing traditional and non-intelligent approaches. For advanced combinatorics problems, intelligent computing becomes 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. Having described how to implement the basic combinatorics problem in Prolog, we will briefly explain how some combinatorics problems may require intelligent computing, particularly in terms of logic programming. While the concept can be applied to any combinatorics problems, we select well known, computationally hard examples.

5.1 Permutations

The traveling salesman problem (TSP) is an NP-complete, famous optimization problem for permutations [2]. 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. The TSP has been chosen as a popular bench mark problem to test the effectiveness of many new techniques. More than 1700 related papers on the TSP and its variants have been published during the five years preceding 2002 [3]. The current techniques can be divided into two categories. One is the 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 [8]. Typically optimal solutions are not guaranteed for the techniques in the second category. For either category, particularly for the first, generation of permutations is 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. There may be preferred sequences of work points because, e.g., it may be more efficient to perform the same type of work consecutively. For example, if City *C* is followed by *D*, then *E* and *F* must or must not follow *D*, and so on. If these conditions are relatively simple, one might be able to implement them by manipulating the distance matrix. But when the conditions become complex, it would be impossible to solve the problem by simply manipulating the distance matrix. There may be different priorities on the points because their urgencies are different, e.g., emergency calls, minor repairs, and routine maintenance work. For such a problem, logic programming will be a powerful tool. These conditions can be expressed in terms of Prolog and imbedded in the permutation program discussed earlier. For example, the condition: "if *C* is followed by *D*, then *E* and *F* must not follow *D*" may be expressed in form of "*C*, *D*, NOT (*E*, *F*)." Prolog implementation of the TSP has been discussed in the literature [1], [4], [10]. Le gives a Prolog program that determines a permutation of cities to be visited and an optimal total distance given a distance matrix. The types of additional conditions discussed above such as preferred sequences can be incorporated into such a program.

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.

5.2 Combinations

Although, the Prolog program for combinations is not presented in this paper, it can be written similarly. 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. It turns out that two other popular problems, the maximum clique and vertex cover, are different versions of essentially the same problem [2]. 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; information retrieval, experimental design, signal transmission, and computer vision; labeled pattern matching; PLA folding; and stereo vision correspondence [9]. As in the cases of the hard permutation problems discussed above, logic programming can be an effective tool for extensions of these combination problems.

6 Conclusion

Combinatorics problems, such as permutations and combinations, have extensive applications and have mostly been studied by classical methods. This article suggests intelligent computing approaches for advanced combinatorics problems by employing logic programming. The approaches may involve processes such as inference and the use of background knowledge. Future studies include actual implementations and comprehensive experiments of these new systems.

References

[1] Bratko I., 2001, *Prolog Programming for Artificial Intelligence*, 3rd, Ed., Addison-Wesley, Wokingham, England.

[2] Garey M.R., Johnson D.S., 1979, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W.H. Freeman, San Francisco.

[3] Jung S., Moon B.-R., 2002, *Toward minimal restriction of genetic encoding and crossovers for the two-dimensional Euclidean TSP*, IEEE Transactions on Evolutionary Computation, Vol.6, No.6, pp. 557–565.

[4] Le T.V., 1993, *Techniques of Prolog Programming with Implementation of Logical Negation and Quantified Goals*, Wiley, New York.

[5] Liu C.L., 1968, *Introduction to Combinatorial Mathematics*, Computer Science Series, McGraw-Hill, New York, Chapter 1.

[6] Munakata T., 1992, *Notes on implementing sets in Prolog*, Communications of the ACM, Vol.35, No.3, pp. 112–120.

[7] Munakata T., 1998, *Notes on implementing fuzzy sets in Prolog*, Fuzzy Sets and Systems, Vol.98, No.3, pp. 311–317.

[8] Munakata T., 1998, *Fundamentals of the New Artificial Intelligence: Beyond Traditional Paradigms*, Springer-Verlag, New York.

[9] Takefuji Y., 1992, *Neural Network Parallel Computing*, Kluwer Academic, Boston, MA.

[10] WASP (Working group on Answer Set Programming), 2005, WASP-Showcase: Knowledge-based planning,
http://www.kr.tuwien.ac.at/projects/WASP/planning.html