An Analysis of Constraint Functional Logic Programming for Resolving Combinatorial Issues: Implementations and Application

Meenaxi R. Sangam[1] Dr. R. A. Khan[2]

[1]Research Scholar, CMJ University, Shillong, Meghalaya

[2]Professor of Mathematics, CMJ University, Shillong, Meghalaya

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.

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.

We present CFLP(FD), a constraint functional logic programming approach over finite domains (FD) for solving typical combinatorial problems. Our approach adds to former approaches as Constraint Logic Programming (CLP), and Functional Logic- Programming (FLP) both expressiveness and further efficiency by combining combinatorial search with propagation. We integrate FD constraints into the functional logic language TOY. CFL P(FD) programs consist of TOY rules with FD constraints declared as functions. CFLP (FD) seamlessly combines the power of CLP over FD with the higher order characteristics of FLP.

----------♦----------

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.

COMBINATORICS EXECUTION 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 ≤ 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.

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.

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 ≤ 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(nPR). 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).
```

APPLICATION ASPECTS 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.

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).

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).

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).

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.

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.

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).

COMPUTATIONAL MODELS

Constraint logic programming (briefly, CLP) [19] is a programming paradigm that is particularly well suited for encoding combinatorial optimization problems. CLP naturally merges two declarative paradigms: constraint solving and logic programming.

Let Σ be a logic language signature $\Sigma = \langle \mathcal{F}, \mathcal{V}, \Pi \cup \Pi_C \rangle$, where

- $\mathcal{F}$ is a finite set of function and constant symbols
- $\mathcal{V}$ is a denumerable collection of variables
- $\Pi \cup \Pi_C$ is a finite set of predicate symbols, where Π and Π_C are disjoint sets.

A constraint is a first-order formula over $\langle \mathcal{F}, \mathcal{V}, \Pi_C \rangle$. Typically, constraints are con junctions of literals, e.g., $0 < X, X < 3, X + Y \neq 4$. Following the traditional logic programming notation, a comma indicates a conjunction, capital letters denote variables, and the symbol :— denotes the implication $\longleftarrow$. CLP lets a programmer use different classes of constraints and domains to encode problems. For combinatorial problems, it is common to use finite domain constraints. namely arithmetic constraints between arithmetic expressions, where each variable is associated to a finite domain of possible values. In this case the interpretation of variables, expressions, and constraints is over $\mathbb{Z}$.

A CLP program over Σ is a finite set of rules of the form $p(s_1, \ldots, s_n) :- C, q_1(t_1^1, \ldots, t_{n_1}^1), \ldots, q_m(t_1^m, \ldots, t_{n_m}^m)$ where C is a constraint. s_i and t_j^i are $(\mathcal{F}, \mathcal{V})$-terms, and $p, q_1, \ldots, q_m$ are predicate symbols of Π. Observe that a CLP program without constraints is in fact a Prolog program.

In contrast to classic generate-and-test approach of logic programming, CLP usually uses a constraw-and-generate technique in which an initial deterministic phase imposes a number of constraints, then a non-deterministic phase generates/explores the solution space. In the constraint phase, in particular, a finite domain of values is assigned to each of the variables. For instance, the constraint domain ([A, B, C] ,1,5) assigns the set of admissible values {1,2,3,4,5} to the variables A. B. and C. The built-in predicate labeling implements the solution search process. Each time a variable is assigned a value, a deterministic propagation stage is executed, pruning the set of values to be attempted for the other variables. Various options (affecting, for instance. the variable selection criteria, the ordering of the attempted values, etc.) can be used to guide the search. The main structure of a program using this programming style is the following:

```
solve_problem(X1,...,Xn) :-
     constraint(X1,...,Xn),
     labeling([options], X1,...,Xn).
```

A CLP(FD) system executes program according to goals provided by the user, where a goal is a conjunction of literals. Given a program P and a goal $a_1, \ldots, a_k$, the program will determine the instantiations σ of the variables in the goal such that $\forall(a_1, \ldots, a_k)\sigma$ is a logical consequence of P.

EXPERIMENTAL SYSTEM

The experimental studies reported in this paper have been mainly conducted using two CLP(FD) implementations and two ASP solvers. The CLP(FD) programs have been designed for execution by SICStus Prolog 3.11.2 (using the library clpfd) and GNU-Prolog 1.2.16—though the code is general enough to be used on other platforms, such as B-Prolog and ECLiPSe, with minimal syntactic adjustments [37]. The ASP programs have been designed to be processed by lparse, the grounding preprocessor adopted by both the SMODELS (version 2.28) and the CMODELS (version 3.03) systems [36]. The CMODELS system makes use of a SAT solver to compute answer sets—in our experiments we used both the default SAT solver, mChaff [28], and Simo [17].

Some experiments have been performed using the SAT solvers zChaff and RelSat. We do not report here results concerning them—the interested reader is referred to [9] for such results. SAT-based ASP solvers, such as CMODELS, take advantage of the tightness of the ASP programs [20]. In presence of non-tight programs, CMODELS is forced to repeatedly call the SAT solver in order to reach a solution. This is done to discard those models of the program that are not answer sets, trying to avoid the introduction of a potentially exponential number of loop formulae [23].

We focused on well-known computationally-hard problems. Among them: Graph k-coloring, Hamiltonian circuit, Schur numbers, protein structure prediction in a 2D lattice [3], planning in a block world, generalized Knapsack, and code design. While some of the programs have been drawn from the best proposals appeared in the literature, others are novel solutions, developed by the authors for this project—e.g., this is the case for the ASP implementation of the protein structure prediction problem and the planning implementation in CLP(FD).

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.

In this paper, we described an experimental study aimed at comparing the performance of CLP(FD) and ASP on various classes of combinatorial problems. The aim of the study is to provide a better understanding of what makes one paradigm more suitable than the other in solving combinatorial problems.

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.

We have presented CFLP(FD), a functional logic programming approach to FD constraint solving, which we think may be pro_tably applied to solve typical problems in the arti_cial intelligence area. We have shown how FD constraints can be de_ned as functions and therefore integrated naturally on FLP languages. Due to its functional component, CFLP(FD) provides better tools, when compared to CLP(FD), for a productive declarative programming. Due to the use of constraints, the expressivity and capabilities of our approach are clearly superior to both those of the functional and purely constraint program-ming approaches.

REFERENCES

- Dovier, A. Formisano, and E. Pontelli. A comparison of CLP(FD) and
- ASP in tackling hard combinatorial problems.
- Bratko I., 2001, Prolog Programming for Artificial Intelligence, 3rd, Ed., Addison-Wesley, Wokingham, England.
- Combinatorial Pattern Matching;In: Proceedings of Annual Symposiums. Lecture Notes in Computer Science. (1992-2000). Springer, Berlin.
- D. Diaz and P. Codognet. Design and Implementation of the GNU-Prolog System. J. of Functional and Logic Programming, 2001(6), 2001.
- J. Jaffar and M. J. Maher. Constraint Logic Programming: A Survey. J. of Logic Programming, 19/20:503–581, 1994.
- 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.
- Liu, Chung Laung. (1968). Introduction to Combinatorial Mathematics, Computer Science Series, New York, McGraw-Hill, Chapter 1.
- M. Dincbas, II. Simonis, and P. Van Hentenryck. Solving Large Combinatorial Problems in Logic Programming. Journal of Logic Programming, 8(l-2):75—93,1990.
- M. Gelfond and V. Lifschitz. The Stable Model Semantics for Logic Programming. In Proc. of ICLP88, pp. 1070–1080,MIT Press, 1988.
- Munakata T., 1998, Fundamentals of the New Artificial Intelligence: Beyond Traditional Paradigms, Springer-Verlag, New York.
- P. Van Hentenryck. Constraint Satisfaction in Logic Programming. The MIT Press, 1989.
- Roberts, Fred S. (1984). Applied Combinatorics, Englewood Cliffs, New Jersey, Prentice-Hall.