Optimizing Logic Programs with Finite-Domain Constraints *

Nai-Wei Lin
Department of Computer Science
The University of Arizona
Tucson, AZ 85721

July 29, 1993

Abstract

Combinatorial problems are often specified declaratively in logic programs with constraints over finite domains. This paper presents several program transformation and optimization techniques for improving the performance of the class of programs with finite-domain constraints. The techniques include planning the evaluation order of body goals, reducing the domain of variables, planning the instantiation order of variable values, and controlling process granularity and scheduling processes among processors in parallel or distributed systems. The techniques are based on the information about the number of solutions of combinatorial problems. Since the decision versions of many combinatorial problems are NP-complete, if P $\neq$ NP, there is no polynomial time algorithm for computing the number of solutions for combinatorial problems. This paper presents a simple greedy approximation algorithm for estimating the number of solutions for combinatorial problems over finite domains. Based on this simple algorithm, a class of flexible polynomial time algorithms is also developed.

*This work was supported in part by the National Science Foundation under grant number CCR-8901283.

1 Introduction

Many problems arising in artificial intelligence and operations research can be formulated as combinatorial problems. A combinatorial problem can be a constraint satisfaction problem or a combinatorial optimization problem. A constraint satisfaction problem involves a set of n variables, a domain of values, and a set of constraints on the variables. A solution to such a constraint satisfaction problem is an n-tuple of assignments of domain values to variables such that all the constraints in the problem are satisfied. Usually, either a single solution or all solutions are sought. A combinatorial optimization problem can be viewed as a constraint satisfaction problem, which is further subject to an object function (a maximization or minimization function). Familiar problems that can be formulated as combinatorial problems include *boolean satisfiability*: deciding whether a boolean formula is satisfiable; *subgraph isomorphism*: given two graphs, deciding whether one graph is isomorphic to a subgraph of the other one; *graph coloring*: using the fewest number of colors to color a graph so that adjacent vertices have different colors; and so on.

Combinatorial problems are often specified declaratively in logic programs with constraints over finite domains. This paper will examine in particular the class of programs whose constraints involve only comparison operators ($=, \neq, >, \geq, <, \leq$). These logic programs are usually solved by tree search algorithms: backtracking [4, 5, 27, 32], intelligent backtracking [2, 20], forward checking, looking ahead, or branch and bound algorithms [36]. The performance of the tree search algorithms depends heavily on the evaluation order among domain-value generators and among constraints [14, 29], and the instantiation order of domain values to variables, that is, the order in which domain values are generated by generators [11]. For ordering the evaluation among generators and constraints, existing logic programming systems either dynamically order the evaluation of subgoals [2, 4, 5, 20, 27, 36], which will inevitably incur some runtime overhead, or use very primitive information for static ordering [32], which only achieves restricted benefits. For ordering the instantiation of domain values to variables, no existing system supports this feature yet. This paper presents several program transformation techniques for statically planning the subgoal evaluation order and the instantiation order of domain values to variables. These techniques allow automatic transformation of declaratively specified programs into more efficient programs. The techniques are based on the information about the number of solutions of constraint satisfaction problems. Moreover, knowledge about the number of solutions of constraint satisfaction problems can also be applied to reduce the domain of variables, and control process granularity and schedule processes among processors in parallel or distributed systems.

The decision versions of many constraint satisfaction problems, such as boolean satisfiability, subgraph isomorphism, and graph k-colorability, that decide whether there exists a solution satisfying all the constraints in the problem are NP-complete [6, 18]. Thus if P $\neq$ NP, there is no polynomial time algorithm for computing the number of solutions for constraint satisfaction problems. Rivin and Zabih [31] have proposed an algorithm for computing the number of solutions for constraint satisfaction problems by transforming a constraint satisfaction problem into an integer linear programming problem. If the constraint satisfaction problem has n variables and m domain values, and if the equivalent programming problem involves M equations, then the number of solutions can be determined in time $O(nm2^{M-n})$.

We are interested in finding algorithms for computing an upper bound on the number of solutions for constraint satisfaction problems over finite domains of discrete values. This paper presents a simple approximation algorithm based on the greedy method. The basic idea of this algorithm is to reduce recursively a constraint satisfaction problem involving n variables into a constraint satisfaction problem involving $n-1$ variables. Through the reduction, the number of solutions of the constraint satisfaction problem involving $n-1$ variables is made to be an upper bound on the number of solutions of the constraint satisfaction problem involving n variables. Based on this simple algorithm, a class of flexible polynomial time approximation algorithms is also developed. A user can choose the appropriate algorithm according to specific efficiency and precision requirements. These algorithms are particularly useful for programs whose constraints are explicitly expressed as a set of constraints involving comparison operators. In general, it is very difficult to infer nontrivial number of solutions information for this class of programs without considering the net effects of the set of constraints. Estimation of the number of solutions for this class of programs has not been addressed in previous work [9, 28, 33, 38].

The remainder of this paper is organized as follows. Section 2 formalizes the problem of computing the number of solutions for constraint satisfaction problems and reduces this problem into a graph-theoretical problem that counts the number of cliques in a graph. It then derives a nontrivial upper bound on the number of cliques in a graph and presents a simple approximation algorithm for estimating the number of cliques in a graph based on the derived upper bound. It finally investigates a number of methods for improving the estimation precision and gives the experimental performance of the described algorithms. Section 3 presents several program transformation and optimization techniques and provides experimental measurements of applying these techniques to a set of benchmark programs. Finally, Section 4 concludes this paper.

2 Number of Solutions of Constraint Satisfaction Problems

In this paper we only consider the following class of **binary constraint satisfaction problems** (binary CSPs): a binary CSP involves a finite set of variables $x_1, \ldots, x_n$, a finite domain of discrete values $d_1, \ldots, d_m$, and a set of unary or binary constraints on variables, namely, constraints involving only one or two variables. A solution to such a CSP is an n-tuple of assignments of domain values to variables such that all the constraints in the problem are satisfied. The approximation of n-ary CSPs (i.e., with constraints involving n variables) by binary CSPs is discussed in [26]. Hereafter we will assume every CSP is a binary CSP unless otherwise mentioned.

2.1 Number of Solutions and Number of N-cliques

The set of constraints on variables can be represented as a graph $G = (V, E)$, called a **consistency graph**. Each vertex $v_{i,j}$ in V denotes the assignment of a value d_j to a variable x_i, written $x_i \leftarrow d_j$, for $1 \leq i \leq n$ and $1 \leq j \leq m$. There is an edge $\langle v_{p,g}, v_{q,h} \rangle$ between two vertices $v_{p,g}$ and $v_{q,h}$ if the two assignments $x_p \leftarrow d_g$ and $x_q \leftarrow d_h$ satisfy all the constraints involving variables x_p and x_q. The two assignments are then said to be **consistent**. The set of vertices $V_i = \{v_{i,1}, \ldots, v_{i,m}\}$ corresponding to a variable x_i is called the **assignment set** of x_i. The **order** of a consistency graph G is (n, m) if G corresponds to a CSP involving n variables and m domain values. Because two distinct values cannot be assigned to the same variable simultaneously, no pair of vertices in an assignment set are adjacent. Therefore, the consistency graph of a CSP involving n variables is an n-partite graph. As an example, the consistency graph for the problem (the 3-queens problem) with the set of variables $\{x_1, x_2, x_3\}$, the domain $\{1, 2, 3\}$ and the constraints $x_j \neq x_i$ and $x_j \neq x_i \pm (j - i)$, for $1 \leq i < j \leq 3$, is shown in Figure 1.

An **n-clique** of a graph G is a subgraph of G such that it has n vertices and its vertices are pairwise adjacent. Since a solution s to a CSP p involving n variables is an n-tuple of assignments of domain values to variables such that all the constraints in p are satisfied, every pair of assignments in s is consistent. Thus, if the constraints involve only conjunctions, then s corresponds to an n-clique of the consistency graph of p, and the number of solutions of p is equal to the number of n-cliques in the consistency graph of p; if the constraints involve also disjunctions, then the number of solutions of p is bounded above by the number of n-cliques in the consistency graph of p. For instance, because there is no 3-clique in the consistency graph in Figure 1, there exists no solution for the corresponding problem.

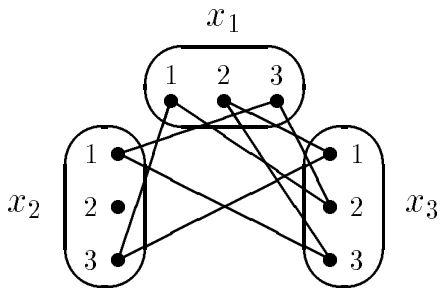

Figure 1: An example of consistency graph

Therefore, the problem of computing an upper bound on the number of solutions of a CSP is reduced to the problem of computing an upper bound on the number of n-cliques in the corresponding consistency graph. We can easily verify that the problem of computing the number of n-cliques in a consistency graph is NP-hard because the enumeration version of the graph coloring problem is NP-hard and can be reduced to this problem. The enumeration version of the graph coloring problem, which involves only conjunctive binary constraints, is the problem that counts the number of different ways of coloring a graph using a given number of colors.

We will use $K(G,n)$ to denote the number of n-cliques in a consistency graph G. Let $G = (V,E)$ be a graph and $N_G(v) = \{w \in V \mid \langle v,w\rangle \in E\}$ be the **neighbors** of a vertex v in G. The **adjacency graph** of v with respect to G, $Adj_G(v)$, is the subgraph of G **induced** by $N_G(v)$, i.e., $Adj_G(v) = (N_G(v), E_G(v))$, where $E_G(v)$ is the set of edges in E that join the vertices in $N_G(v)$. The following theorem shows that the number of n-cliques in a consistency graph can be represented in terms of the number of $(n-1)$-cliques in the adjacency graphs corresponding to the vertices in an assignment set.

Theorem 2.1 *Let G be a consistency graph of order (n,m). Then for each assignment set $V = \{v_1,\ldots,v_m\}$,*

$$K(G,n) = \sum_{i=1}^{m} K(Adj_G(v_i), n-1). \tag{1}$$

Proof Let G_i be the subgraph of G induced by $N_G(v_i) \cup \{v_i\}$. Since no pair of vertices in V are adjacent, $K(G,n) = \sum_{i=1}^{m} K(G_i,n)$. Because v_i is adjacent to every vertex in $N_G(v_i)$, $K(G_i,n) = K(Adj_G(v_i), n-1)$. □

Theorem 2.1 says that the problem of computing the number of n-cliques in a consistency

graph of order (n, m) can be transformed into m subproblems of computing the number of $(n-1)$-cliques in a consistency graph of order $(n-1, m)$. However, when m is larger than 1, the computation will require exponential time $O(m^n)$. To make it practical, therefore, we need to find a way to combine the set of subgraphs $Adj_G(v_1), \ldots, Adj_G(v_m)$ in Formula (1) into a graph H such that $K(H, n-1)$ is an upper bound on $K(G, n)$.

2.2 An Upper Bound on the Number of N-cliques

To derive an upper bound on $K(G, n)$ for a consistency graph G of order (n, m), we extend the representation of a consistency graph to a *weighted* consistency graph. A **weighted consistency graph** $G = (V, E, W)$ is a consistency graph with each edge $e \in E$ associated with a **weight**, $W(e)$, where $W : V \times V \rightarrow N$ is a function that assigns a positive integer to an edge $\langle u, v \rangle$ if $\langle u, v \rangle \in E$, and assigns 0 to $\langle u, v \rangle$ if $\langle u, v \rangle \notin E$. The aim is to use the weights to accumulate the number of n-cliques information.

The number of n-cliques, $K(G, n)$, in a *weighted* consistency graph G of order (n, m) is defined as follows. Let S be the set of n-cliques of G and $H = (V_H, E_H, W_H) \in S$ be an n-clique. We define $K(H, n) = min\{W_H(e) \mid e \in E_H\}$, and $K(G, n) = \sum_{H \in S} K(H, n)$. The intuition behind this formulation involves an operation on graphs and will become clear shortly. Let the weighted consistency graph corresponding to a consistency graph $G = (V, E)$ be $G' = (V, E, W)$, where $W(e) = 1$, for all $e \in E$. Then we have $K(G, n) = K(G', n)$ by definition. Thus we can focus on weighted consistency graphs from now on.

We now define a binary operator $\oplus$, called **graph addition**, on two weighted consistency graphs. Let $G_1 = (V, E_1, W_1)$ and $G_2 = (V, E_2, W_2)$ be two weighted consistency graphs with the same set of vertices. Then $G_1 \oplus G_2 = (V, E_{1\oplus 2}, W_{1\oplus 2})$, where $E_{1\oplus 2} = E_1 \cup E_2$, and $W_{1\oplus 2}(e) = W_1(e) + W_2(e)$, for all $e \in E_{1\oplus 2}$. Graphs G_1 and G_2 are said to be the **component graphs** of $G_1 \oplus G_2$. An example of graph addition is shown in Figure 2. We use the same notation as the one in Figure 1. Suppose every edge in the graphs G_1 and G_2 has weight 1. We can easily verify that every edge in the graph $G_1 \oplus G_2$ has weight 1 except for edges $\langle 12, 21 \rangle$, $\langle 12, 33 \rangle$ and $\langle 21, 33 \rangle$, which have weight 2, where $\langle ij, gh \rangle$ denotes the edge joining the vertices that represent $x_i \leftarrow d_j$ and $x_g \leftarrow d_h$.

The intuition behind the definition of the number of n-cliques in a weighted consistency graph G is that for each n-clique H of G, $K(H, n) = k$ implies that H appears in *at most* k component graphs of G. For instance, for the consistency graph $G_1 \oplus G_2$ in Figure 2, $G_1 \oplus G_2$ has three 3-cliques $\langle 11, 23, 32 \rangle$, $\langle 12, 21, 33 \rangle$ and $\langle 13, 22, 31 \rangle$, and 3-clique $\langle 12, 21, 33 \rangle$ appears in

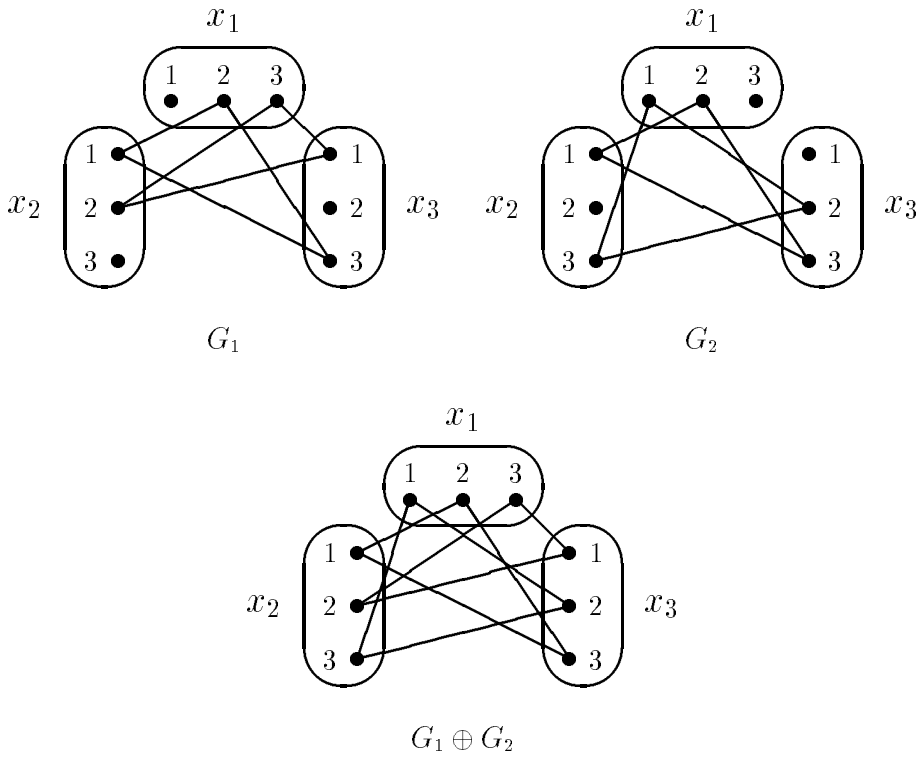

Figure 2: An example of graph addition

both G_1 and G_2. Thus $K(G_1 \oplus G_2, 3) = 1+2+1 = 4$. The following theorem shows the effect of graph addition on the number of n-cliques in the graphs.

Theorem 2.2 *Let $G_1 = (V, E_1, W_1)$ and $G_2 = (V, E_2, W_2)$ be two weighted consistency graphs of order (n, m). Then*

$$K(G_1 \oplus G_2, n) \geq K(G_1, n) + K(G_2, n). \tag{2}$$

Proof Let S, S_1 and S_2 be the sets of n-cliques of $G_1 \oplus G_2, G_1$ and G_2 respectively. Then $S = S_1 \cup S_2$. Let $H = (V_H, E_H, W_H) \in S$ be an n-clique. Then $W_H(e) = W_1(e) + W_2(e)$, for all $e \in E_H$. If H is in both G_1 and G_2, then $min\{W_H(e) \mid e \in E_H\} \geq min\{W_1(e) \mid e \in E_H\} + min\{W_2(e) \mid e \in E_H\}$. If H is in either G_1 or G_2, but not both, then $min\{W_H(e) \mid e \in E_H\} \geq min\{W_1(e) \mid e \in E_H\}$ and $min\{W_2(e) \mid e \in E_H\} = 0$, or $min\{W_H(e) \mid e \in E_H\} \geq min\{W_2(e) \mid e \in E_H\}$ and $min\{W_1(e) \mid e \in E_H\} = 0$. If H is in neither G_1 nor G_2, then $min\{W_H(e) \mid e \in E_H\} > 0$, and $min\{W_1(e) \mid e \in E_H\} = min\{W_2(e) \mid e \in E_H\} = 0$. □

Theorem 2.2 shows that graph addition is a way to combine the set of adjacency graphs so that the number of n-cliques in the combined graph is an upper bound on the sum of the

number of n-cliques in each individual adjacency graph. We now define the weight of an edge in a weighted adjacency graph as follows. Let $Adj_G(v) = (V_A, E_A, W_A)$ be the weighted adjacency graph of v with respect to a weighted consistency graph $G = (V, E, W)$. For every edge $\langle u, w\rangle \in E_A$, we define $W_A(\langle u, w\rangle) = min(W(\langle v, u\rangle), W(\langle v, w\rangle), W(\langle u, w\rangle))$. This definition will lead to the same result for weighted consistency graph as Theorem 2.1 for consistency graph.

Theorem 2.3 *Let G be a weighted consistency graph of order (n, m). Then for each assignment set $V = \{v_1, \ldots, v_m\}$,*

$$K(G, n) = \sum_{i=1}^{m} K(Adj_G(v_i), n-1). \tag{3}$$

Proof Let $G_i = (V_{G_i}, E_{G_i}, W_{G_i})$ be the subgraph of G induced by $N_G(v_i) \cup \{v_i\}$. Since no two vertices in V are adjacent, $K(G, n) = \sum_{i=1}^{m} K(G_i, n)$. Let $Adj_G(v_i) = (V_{A_i}, E_{A_i}, W_{A_i})$. Because v_i is adjacent to every vertex in $N_G(v_i)$, $min\{W_{G_i}(e) \mid e \in E_{G_i}\} = min\{W_{A_i}(e) \mid e \in E_{A_i}\}$. Therefore, $K(G_i, n) = K(Adj_G(v_i), n-1)$. □

Thus we have the following theorem to conclude the derivation of an upper bound on the number of n-cliques in a weighted consistency graph.

Theorem 2.4 *Let G be a weighted consistency graph of order (n, m). Then for each assignment set $V = \{v_1, \ldots, v_m\}$,*

$$K(G, n) \leq K(\bigoplus_{i=1}^{m} Adj_G(v_i), n-1). \tag{4}$$

Proof By Theorems 2.2 and 2.3. □

2.3 An Approximation Algorithm

We are now ready to present a simple greedy algorithm for computing an upper bound on $K(G, n)$ for a weighted consistency graph G of order (n, m). The basic idea is to apply Theorem 2.4 repeatedly to a sequence of consecutively smaller graphs. By starting with the graph G, at each iteration, one assignment set is removed from the graph, and a smaller graph is constructed by performing graph addition on the set of adjacency graphs corresponding to the vertices in the removed assignment set. This assignment set elimination process continues until there are only two assignment sets left. The resultant graph is now a bipartite graph. By

definition, the number of 2-cliques in a weighted bipartite consistency graph is the sum of the weights of the edges (2-cliques) in the graph. The algorithm is shown as follows:

Algorithm n-cliques$(G = (V, E, W), n, m)$
1. **begin**
2. $G_1 := G$;
3. **for** $i := 1$ **to** $n - 2$ **do**
4. $G_{i+1} := \bigoplus_{j=1}^{m} Adj_{G_i}(v_{i,j})$;
5. **od**
6. **return** $\sum_{e \in E_{n-1}} W_{n-1}(e)$;
7. **end**

Let us consider the time complexity of the n-cliques algorithm. Let $v = nm$ be the number of vertices in the graph and e the number of edges in the graph. Each adjacency graph can be constructed in time $O(v + e)$, and each graph addition can be performed in time $O(v + e)$. Therefore, the total time required for Line 4 is $O(m(v + e))$. Taking the loop into account, Lines 3-5 require time $O(nm(v + e)) = O(v(v + e))$. The summation of weights for a bipartite graph in Line 6 can be performed in time $O(m^2)$. Thus the time complexity for the entire n-cliques algorithm is $O(v(v + e))$.

2.4 Improving Estimation Precision

In this subsection we investigate three methods for improving the estimation precision of the n-cliques algorithm: (1) **network consistency**; (2) **exact expansion**; and (3) **assignment memorization**. For all three methods, applying in full extent will lead to exact solution and exponential time. Therefore, we will examine the tradeoff between efficiency and precision of these methods.

2.4.1 Network Consistency

Since basic backtracking algorithms may incur significant inefficiency, a class of network consistency algorithms has been proposed to improve the efficiency of backtracking algorithms [12, 13, 14, 22, 25, 26, 37]. The basic idea behind the network consistency algorithms is as follows. Each individual constraint in a CSP only makes the local consistencies (consistent assignments between two variables) explicit. Through exploiting some global consistencies (i.e.,

consistent assignments among more than two variables), we might remove beforehand some of the domain values from consideration at each stage of a backtracking algorithm.

We can use the same idea to reduce the consistency graph by removing the edges (local consistencies) that are unable to satisfy global consistencies. For example, a vertex v in an assignment set must be adjacent to a vertex in every other assignment set so that the assignment corresponding to v may be in a potential solution; otherwise, we can remove all the edges incident on v. A graph is said to be **2-consistent** if all its vertices satisfy the above condition. More generally, a consistency graph G is said to be **k-consistent** if for every $(k-1)$-clique $H = (V, E, W)$ in G, there exists at least one vertex v in every assignment set, apart from those assignment sets containing the vertices in V, such that the subgraph induced by $V \cup \{v\}$ is a k-clique. To incorporate an algorithm k-consistency, which achieves k-consistency of a graph, into the n-cliques algorithm, we can just replace the formula $\bigoplus_{j=1}^{m} Adj_G(v_{i,j})$ in the n-cliques algorithm by formula $\bigoplus_{j=1}^{m} k\text{-consistency}(Adj_G(v_{i,j}))$. The time complexity for the network consistency algorithms that achieve 2-consistency and 3-consistency are $O(n^2m^3)$ and $O(n^3m^5)$ respectively [23].

2.4.2 Exact Expansion

The exact expansion method tries to balance the precision of Formula (3) and the efficiency of Formula (4). The intent is to first expand the Formula (3) some number of times to exactly generate some subproblems, then use the Formula (4) to approximately solve the expanded subproblems. Each time the Formula (3) is used, the time complexity of the entire algorithm will increase by a factor of $O(m)$.

2.4.3 Assignment Memorization

In the n-cliques algorithm, the weight of an edge e at the end of the i^{th} iteration denotes the number of component graph sequences $A_1, \ldots, A_i$ in which the edge e occurs, where A_j is the component graph in which the edge e occurs at the j^{th} iteration. Or equivalently, it denotes the number of partial assignments to variables $x_1, \ldots, x_i$ with which the two assignments corresponding to the edge e are consistent. Due to the lack of partial assignment information, graph addition is performed without knowing to which partial assignments the weight contributes. The assignment memorization method tries to memorize the weight as well as some partial assignment information so that we can take advantage of this information and perform graph addition in a more precise way.

Operationally, to memorize the most recent variable assignment, each edge of the weighted consistency graph needs to maintain an array of m weights instead of a single weight. At the end of the i^{th} iteration of the n-cliques algorithm, the j^{th} element of the weight array of an edge e, $W(e)[j]$, corresponds to the number of partial assignments to variables $x_1, \ldots, x_i$ to which e contributes with $x_i \leftarrow d_j$. Therefore, let $G_{i+1} = (V, E, W)$ be the graph at the end of the i^{th} iteration, and $Adj_{G_i}(v_{i,j}) = (V_j, E_j, W_j)$ be the j^{th} adjacency graph at the i^{th} iteration. For each edge $e \in E$, we have $W(e)[j] = \sum_{k=1}^{m} W_j(e)[k]$. The memorization of k variable assignments will cost the entire algorithm a factor of $O(m^k)$ in both time and space.

2.5 Experimental Performance

We now give the experimental performance of the algorithms described in the previous subsections. Experiments are performed on a set of randomly generated consistency graphs. The edges in the graphs are chosen independently and with the probability 0.75. The probability 0.75 is chosen so that the graphs have reasonably high density to have a reasonable number of cliques.

Apart from the n-cliques algorithm, we also incorporate network consistency, exact expansion and assignment memorization algorithms into the n-cliques algorithm. The 2-consistency and 3-consistency algorithms achieve respectively 2-consistency and 3-consistency for each constructed adjacency graph; the 1-expansion and 2-expansion algorithms expand the Formula (3) once and twice respectively; the 1-memorization and 2-memorization algorithms memorize respectively the most recent one and two variable assignments. We apply each algorithm to 16 random graphs for each of the orders (4,4), (5,5), ..., (8,8).

The performance results are shown in Figure 3. Figures 3(a), 3(c) – 3(h) display the relative error of the algorithms with respect to the order of the consistency graphs. The relative error is defined as $(K - K^*)/K^*$, where K and K^* denote, respectively, the estimated and the exact number of n-cliques in the graph. The results show that the relative error of the algorithms increases as the order of the graph grows. Because the number of graph additions performed increases as the order of the graph grows, the estimation error accumulated via graph addition also increases. Furthermore, since the number of n-cliques in a graph usually grows exponentially with respect to the order of the graph, the growth rate of the relative error is also very high. However, the relation between two exact solutions usually also reflects upon the corresponding estimated solutions. This result is shown in Figure 3(b). Here we compare the relation between two exact solutions with the relation between the two corresponding estimated solutions for all pairs of the 16 random graphs. The relativity ratio is the ratio

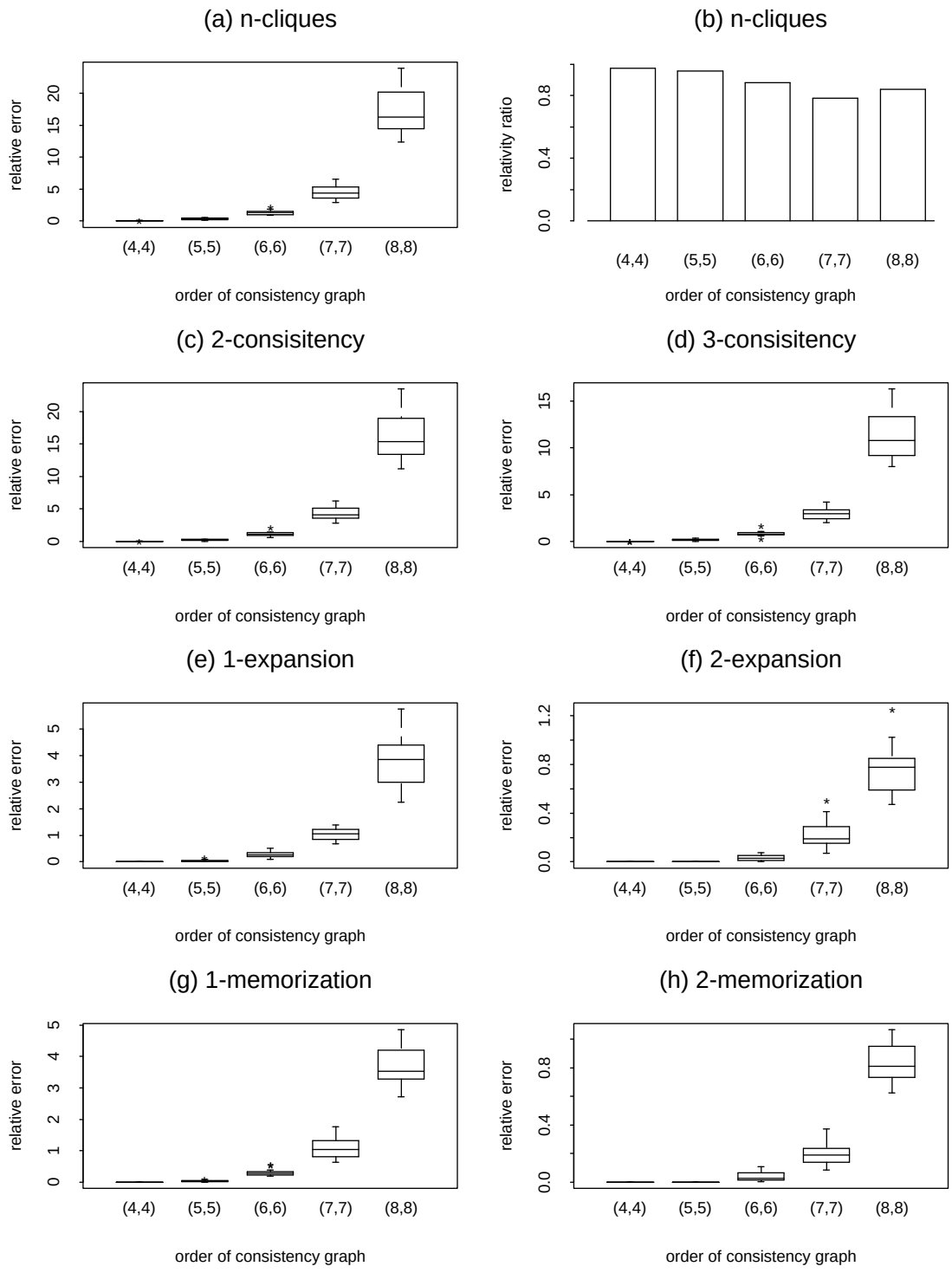

Figure 3: Experimental performance

between the number of pairs that preserve the relation and the total number of pairs.

The results also show that the low-order network consistency algorithms do not work well for dense graphs. In general, as the density of graphs becomes higher, the graphs are more likely to be low-order consistent. However, for applications where network consistency algorithms work well for finding solutions, the method of using network consistency algorithms for estimating the number of solutions will also work well. On the other hand, the exact expansion algorithms and the assignment memorization algorithms can still significantly improve the estimation precision for dense graphs. Moreover, the relative error growth rate for the assignment memorization algorithms is slightly more stable than for the exact expansion algorithms. The reason is that the assignment memorization algorithms improve the precision evenly at each iteration of the process, while the exact expansion algorithms improve the precision mainly at the beginning iterations of the process.

Note also that because these algorithms can achieve any degree of precision by paying the price on efficiency, it is also possible to construct adaptive algorithms that will adaptively decide the amount of efforts to invest based on the order and density of the graphs and the precision requirements.

3 Transformation and Optimization Techniques

The algorithms presented in the previous section are particularly useful for programs whose constraints are explicitly expressed as a set of constraints involving comparison operators. In general, it is very difficult to infer nontrivial number of solutions information for this class of programs without considering the net effects of the set of constraints. However, given the domain information about the variables in the constraints,[1] the consistency graph corresponding to the constraints can be efficiently built by using simple integer interval arithmetic and set manipulations. Given the consistency graph, nontrivial number of solutions information can be efficiently inferred as well by using the presented algorithms. This section uses an example to illustrate several program transformation and optimization techniques based on the number of solutions information, and gives experimental measurements on a set of benchmark programs. The measurements are conducted using SICStus Prolog [3] and running on Sun4.

The benchmark programs include a crypt-arithmetic problem: the send-more-money puzzle [21]; the eight-queens problem; the five-houses puzzle [21]; a job scheduling problem [1, 36]; a

[1] Domain information can be inferred at compile time [15, 24, 30, 39] or declared by users [36].

boolean satisfiability problem: a liar puzzle [21]; the magic 3×3 squares problem [21]; a map coloring problem [2]; and a network flow problem [34].

The example problem is a job scheduling problem [1, 36]. The problem is to find schedules for a project that satisfy a variety of constraints among its jobs. Typical constraints in job scheduling problems include *precedence constraints*, which specify the precedence among the jobs; *disjunctive constraints*, which indicate the mutual exclusion among the jobs due to the sharing of resources; and many other constraints. An example of precedence constraints is given by the following predicate:

```
precedence_constraints(SA,SB,SC,SD,SE,SF,SG,SEnd) :-
    SB >= SA + 2, SC >= SA + 2, SD >= SA + 2, SE >= SB + 3, SE >= SC + 5,
    SF >= SD + 6, SG >= SE + 2, SG >= SF + 3, SEnd >= SG + 1.
```

Here variables `SA`,...,`SEnd` represent the starting time of the jobs $A, \ldots, End$. The constraint '`SB >= SA + 2`' specifies that the job A takes 2 units of time to complete and the job B can be started only after the job A has completed. The job End is a dummy job denoting the end of the project. An example of disjunctive constraints is given by the following predicate:

```
disjunctive_constraints(SB,SC) :- SC >= SB + 3.
disjunctive_constraints(SB,SC) :- SB >= SC + 5.
```

This predicate indicates that the jobs B and C cannot be performed at the same time due to the sharing of resources. We can then express the constraints for the project that involves the above precedence and disjunctive constraints as follows:

```
schedule_constraints(SA,SB,SC,SD,SE,SF,SG,SEnd) :-
    precedence_constraints(SA,SB,SC,SD,SE,SF,SG,SEnd),
    disjunctive_constraints(SB,SC).
```

Given the duration in which the project is supposed to be carried out, a schedule that satisfies the schedule constraints can be expressed as follows:

```
schedule(Duration,SA,SB,SC,SD,SE,SF,SG,SEnd) :-
    generator(Duration,SA), generator(Duration,SB),
    generator(Duration,SC), generator(Duration,SD),
    generator(Duration,SE), generator(Duration,SF),
```

```
    generator(Duration,SG), generator(Duration,SEnd),
    schedule_constraints(SA,SB,SC,SD,SE,SF,SG,SEnd).
```

Here the predicate `generator/2` generates all possible time slots (integers in this case) in a given duration.

3.1 Ordering Subgoals

Seki and Furukawa [32] have presented a technique of transforming a generate and test program into a more efficient program by interleaving generators and constraints using the mode information (specifying the input and output characterization of arguments) of the generators. In their method, subgoals are unfolded and rearranged so that constraints are interleaved into generators immediately after the constraints become active. Using this technique, the predicate `schedule/9` can be transformed into the following more efficient predicate:

```
schedule(Duration,SA,SB,SC,SD,SE,SF,SG,SEnd) :-
    generator(Duration,SA), generator(Duration,SB), SB >= SA + 2,
    generator(Duration,SC), SC >= SA + 2, disjunctive_constraints(SB,SC),
    generator(Duration,SD), SD >= SA + 2,
    generator(Duration,SE), SE >= SB + 3, SE >= SC + 5,
    generator(Duration,SF), SF >= SD + 6,
    generator(Duration,SG), SG >= SE + 2, SG >= SF + 3,
    generator(Duration,SEnd), SEnd >= SG + 1.
```

Notice that the rearrangement of the constraints is based on the order of the generators, and the order of the generators are usually unchanged. However, an appropriate rearrangement of generators may often significantly improve the program efficiency as well. We now examine two program transformation techniques that extend the technique of Seki and Furukawa by also rearranging the order of generators. These techniques are based on the *fail-first principle* [14]. This principle advices to try early the part that is most likely to fail so that the pruning can be performed early. The first transformation technique is based on global number of solutions information, while the second one is based on local number of solutions information.

3.1.1 Global Number of Solutions Information

Applying the fail-first principle, we can use the heuristic that chooses early the generators that generate the fewest values satisfying the constraints in the program. The n-cliques algorithm

can be employed to estimate this kind of global number of solutions information associated with each variable. In essence, the 1-expansion n-cliques algorithm is applied to each variable so that we can obtain separately the number of solutions associated with each domain value of the variables. Since the n-cliques algorithm is an upper bound estimation, for each variable we have an upper bound estimate on the number of domain values satisfying the set of constraints in the program. As a result, for a consistency graph involving v vertices and e edges, computing the number of solutions for all the variable values (i.e., all the vertices in the graph) will require the time complexity $O(v^2(v+e))$.

Applying the 1-expansion n-cliques algorithm to the predicate `schedule_constraints`/8 in the example program for the duration `[1..15]`, the number of solutions associated with each variable value is estimated as follows. We use a pair (i, j) related to a variable v to designate that there are j solutions in which the value of v is i. If there is no entry for a value i, then no solution is associated with i.

`SA`: (1,60), (2,6);
`SB`: (3,24), (4,11), (8,18), (9,15);
`SC`: (3,24), (4,9), (6,18), (7,15);
`SD`: (3,32), (4,24), (5,10);
`SE`: (11,24), (12,42);
`SF`: (9,12), (10,26), (11,28);
`SG`: (13,12), (14,54);
`SEnd`: (14,6), (15,60).

Therefore, in the example, variable `SA` has the value 1 in 60 solutions, the value 2 in 6 solutions, and no other values of `SA` can satisfy the set of constraints in the program. Based on the number of solutions information, we can assign an initial order among the variables as {`SA`, `SE`, `SG`, `SEnd`, `SD`, `SF`, `SB`, `SC`}. Nevertheless, this initial order is not necessary to work well because we still have to interleave the generators and the constraints appropriately. For example, note that in predicate `precedence_constraints`/8 there is no constraint involving both variables `SA` and `SE`. If we follow the initial order, in order to start executing the constraints, we still need to generate values for another variable. We will generate values for variable `SD` if we want to test the constraint '`SD >= SA + 2`' first, or for variable `SC` if we want to test the constraint '`SE >= SC + 5`' first. In the former case, there is no need to generate values for `SE` before the values of `SD` are generated; on the other hand, in the latter case, it is not necessary to put the generator for `SA` before the generators for `SC` and `SE`.

We now describe a simple scheme for adjusting the initial order among variables by taking into account the relationships of variables in the constraints. We call a constraint c a *forward-checkable constraint* of a variable v if v appears in c and all the other variables in c are ordered ahead of v [36]. That is, a forward-checkable constraint of v will become active when v is instantiated. We will associate each variable v with two lists: a variable list that contains the variables appearing together with v in at least one constraint, and a constraint list that contains the forward-checkable constraints of v according to a partially determined ordering. For example, since variable `SB` appears with variable `SA` in the constraint '`SB >= SA + 2`', `SB` is included in the variable list associated with `SA`. In addition, if `SB` has been ordered ahead of `SA`, then the constraint '`SB >= SA + 2`' becomes a forward-checkable constraint of `SA` and is included in the constraint list associated with `SA`. Thus for the example problem we have the following association at the beginning:

`SA`: {`SB, SC, SD`}, {}
`SB`: {`SA, SC, SE`}, {}
`SC`: {`SA, SB, SE`}, {}
`SD`: {`SA, SF`}, {}
`SE`: {`SB, SC, SG`}, {}
`SF`: {`SD, SG`}, {}
`SG`: {`SE, SF, SEnd`}, {}
`SEnd`: {`SG`}, {}

To rearrange the order, we start with choosing the variable `SA` as the first variable because it is the variable with the fewest number of values satisfying the constraints in the program. After variable `SA` is chosen as the first variable to generate, the association becomes

`SB`: {`SC, SE`}, {`SB >= SA + 2`}
`SC`: {`SB, SE`}, {`SC >= SA + 2`}
`SD`: {`SF`}, {`SD >= SA + 2`}
`SE`: {`SB, SC, SG`}, {}
`SF`: {`SD, SG`}, {}
`SG`: {`SE, SF, SEnd`}, {}
`SEnd`: {`SG`}, {}

Notice the updates of the constraint lists associated with variables `SB, SC` and `SD`. Then among the variables in the list associated with `SA`, i.e., {`SB,SC,SD`}, we choose the next variable according to the following criteria:

1. choose the variable with the fewest number of values satisfying the constraints;
2. if there is a tie, choose the variable with the most number of forward-checkable constraints;
3. if there is still a tie, use the natural order of generators.

In this case, it is variable SD. Thus at this point, the constraint 'SD >= SA + 2' can be tested right after the values of SA and SD are generated. After that, the variables in both the list associated with SA and the list with SD, namely, {SB,SC,SF}, are under consideration. Among them, variable SF would be chosen as the next variable. We can continue this process and generate the new order as {SA,SD,SF,SG,SE,SEnd,SB,SC}. According to this new order, an even more efficient predicate schedule/9 can be given as follows:

```
schedule(Duration,SA,SB,SC,SD,SE,SF,SG,SEnd) :-
    generator(Duration,SA), generator(Duration,SD), SD >= SA + 2,
    generator(Duration,SF), SF >= SD + 6,
    generator(Duration,SG), SG >= SF + 3,
    generator(Duration,SE), SG >= SE + 2,
    generator(Duration,SEnd), SEnd >= SG + 1,
    generator(Duration,SB), SB >= SA + 2, SE >= SB + 3,
    generator(Duration,SC), SC >= SA + 2, SE >= SC + 5,
    disjunctive_constraints(SB,SC).
```

We have conducted experimental measurements on the benchmark programs. The programs are executed for a generate and test version (GT), a version using the transformation technique of Seki and Furukawa (SF), and a version using our goal ordering transformation technique based on global number of solutions information (GO). The result is shown in the Table 1. The result shows that the goal ordering transformation based on the local number of solutions information produces more efficient program than Seki and Furukawa's technique for all the benchmark programs except the liar puzzle program. The reason for the worse performance of the GO version of the liar puzzle program is as follows. The 1-expansion n-cliques algorithm infers that there is only one domain value satisfying the constraints in the program for every variable. Therefore, the ordering decision is solely based on the number of forward-checkable constraints associated with each variable. However, in the presented simple scheme, different types of constraints are not distinguished between themselves. We can easily see that equalities usually prune more search space than inequalities, and they should have heavier weight than

Benchmarks	GT	SF	GO	GO/SF	LO	LO/SF
send-more-money	3600.000 ↑	34.6400	0.0781	0.0022	0.0828	0.0023
eight-queens	2496.650	0.9879	0.9879	1.0000	0.8419	0.8522
five-houses	3600.000 ↑	0.3451	0.1382	0.4004	0.0409	0.1185
job-scheduling	3600.000 ↑	4.2800	0.7531	0.1759	0.6109	0.1427
liar-puzzle	0.0659	0.0067	0.0068	1.0149	0.0066	0.9850
magic-squares	3600.000 ↑	0.8469	0.2322	0.2741	0.2132	0.2517
map-coloring	2608.700	3.4029	0.8948	0.2529	1.0014	0.2942
network-flow	1595.650	0.7855	0.6649	0.8464	0.6649	0.8464

Table 1: Measurements for goal ordering transformations (times in seconds)

inequalities. The worse performance of the GO version of the liar puzzle program is mainly due to this kind of imprecision. This drawback will be alleviated in the technique using local number of solutions information.

3.1.2 Local Number of Solutions Information

Instead of using the number of solutions information subject to all the constraints, we now use the number of solutions information subject to each individual constraint. Besides, this information will be applied locally. Applying the fail-first principle, we will use the heuristic that chooses early the generators that generate the fewest values satisfying the *active* constraints instead of all the constraints in the program. Given the domain information for each variable, we will estimate the success probability of each constraint, and estimate the joint success probability of a set of active constraints. We will assume that a domain is given as an integer interval or a set of atoms so that we can use simple integer interval arithmetic for arithmetic constraints and simple set manipulations for non-arithmetic equality or disequality constraints.

Since the set of constraints that become active after a variable v is instantiated is just the set of forward-checkable constraints of v with respect to the set of variables instantiated ahead of v, we will estimate the success probability of a constraint under the condition that it is forward-checkable. As a result, for each constraint c, we can associate a (possibly distinct) success probability of c with each variable occurring in the constraint because any ordering

of variables is possible. For instance, let us consider the constraint '`SB >= SA + 2`' in the example problem. Given the duration `[1..15]`, if `SA` is ordered ahead of `SB`, then we can view the constraint as '`SB >= [1..15] + 2`'. Using integer interval arithmetic, we can obtain '`SB >= [3..17]`'. If we assume that all the values in `[3..17]` have the same distribution, then the success probability of the constraint is 0.404 because there are totally 15×15 distinct `(SA,SB)` value pairs and $(13 \times 14)/2$ of them satisfy the constraint. We make the assumption that all the values in the computed interval have the same distribution for the sake of efficiency. It would be too expensive to maintain distribution information for each individual value in the interval. Similarly, we can obtain '`SA =< [-1..13]`' if `SB` is ordered ahead of `SA`. Assuming that all the values in `[-1..13]` have the same distribution, the success probability of the constraint is also 0.404. However, in general, the success probability of a constraint may be different for different variables. In this way, we can estimate the success probability for each constraint with respect to each variable occurring in the constraint.

We now consider the joint success probability of a conjunction or disjunction of constraints. Let $c_1, \ldots, c_n$ be a conjunction of forward-checkable constraints of a variable v with the success probabilities $p_1, \ldots, p_n$. If we assume that the success probabilities of the constraints are independent of each other, then the joint success probability of this set of constraints can be estimated simply as $\prod_{i=1}^{n} p_i$. We also make the assumption that the success probabilities of the constraints are independent of each other for the sake of efficiency. Maintaining the dependency between the constraints will complicate the analysis significantly. Similarly, let $c_1, \ldots, c_n$ be a disjunction of forward-checkable constraints of a variable v with the success probabilities $p_1, \ldots, p_n$. Assuming that the success probabilities of the constraints are independent of each other, the joint success probability of this set of constraints can be estimated simply as $1 - \prod_{i=1}^{n}(1 - p_i)$.

We now describe a simple scheme for ordering the variables for instantiation. For each variable v, we maintain two lists of constraints: a list that contains the forward-checkable constraints of v, and a list that contains the constraints in which v appears but which is not forward-checkable. The first list of constraints has an immediate effect on the search space when v is instantiated, while the second list of constraints has impact at the later stages of the search. Let n be the domain size of v, and p be the joint success probability of the set of forward-checkable constraints of v, namely, the first list of constraints associated with v. Then the number of values of v satisfying the set of forward-checkable constraints can be estimated as np. It is interesting to note that in the context of forward checking algorithm this estimate can be interpreted as the number of values left in the domain of v. Simultaneously, we also estimate the joint success probability of the constraints in the second list associated with v.

Let this probability be q. Then we will use the following criteria to order the variables:

1. choose the variable with the fewest number of values satisfying its forward-checkable constraints, i.e., with the smallest value of np;
2. if there is a tie, choose the variable with the smallest success probability q (or the largest failure probability $1 - q$);
3. if there is still a tie, use the natural order of generators.

Apart from the rearrangement of the generators, using the success probabilities of the forward-checkable constraints, we can also rearrange the order of the active constraints. Applying the fail-first principle, we will test early the constraints that have the smallest success probability so that fewer tests will be performed.

Experimental measurements for the goal ordering transformation based on local number of solutions information have also been conducted. The version using the local number of solutions information (LO) is compared with the version using the technique of Seki and Furukawa (SF). The result is also shown in the Table 1. The result shows that the goal ordering transformation based on the local number of solutions information produces more efficient program than Seki and Furukawa's technique for all the benchmark programs. The result also shows that there is no clear winner for the two transformation techniques based on the number of solutions information. For the send-more-money program and the map coloring program, the technique using global number of solutions information is better. On the other hand, for other programs, the technique using local numer of solutions information is superior. We also note that for the eight-queens program, although the order of the generators are the same in all three versions, the better performance of the LO version is due to the rearrangement among the active constraints.

Note that the above simple scheme ignores the different costs among generators and constraints. It is possible to take into account the different costs of generators and constraints when making ordering decision. For example, we have used the number of procedure calls as the metric of cost to estimate the different costs of generators and constraints. For any two variables v_1 and v_2, assume that v_1 is ordered ahead of v_2. Then let t_1 be the cost for executing the generator for v_1 and the set of forward-checkable constraints of v_1, and t_2 be the cost for executing the generator for v_2 and the set of forward-checkable constraints of v_2. Also, let n be the domain size of v_1, and p be the joint success probability of the set of forward-checkable constraints of v_1. If these two variables are consecutively ordered, then the cost of executing

them would be $t_1 + npt_2$. Similarly, we can also estimate the cost for the case when v_2 is ordered ahead of v_1. With these cost information, we will choose to use the order that produces the smaller cost.

We have also performed experimental measurements on the benchmark programs for this extended goal ordering transformation technique. However, we obtain the same variable orders as the ones produced by the transformation technique using local number of solutions information for all the benchmark programs. Besides, there are only a few differences on the order among active constraints. Therefore, we do not include this set of measurements in the paper. Nevertheless, we expect that we can certainly benefit from this extended technique for programs with more complicated constraints.

In constraint programming language CHIP [36], consistency techniques (e.g., forward checking and looking ahead) are used to solve combinatorial problems. To apply consistency techniques, the domains of variables are maintained at runtime so that inconsistent domain values can be eliminated actively through the propagation of constraints. With the domain information available at runtime, many heuristics for ordering variables for instantiation can be performed dynamically. For example, using the fail-first principle, the variable with the fewest values left in the domain is chosen as the next variable to instantiate. This dynamic ordering can improve the program efficiency in many cases. However, due to the runtime overhead incurred by dynamic ordering, in some cases applying only consistency techniques is better than applying both consistency techniques and dynamic ordering [7, 36]. In such cases static ordering provides an alternative approach to improving the program performance.

3.2 Reducing Variable Domains

Since the estimation of the n-cliques algorithm is an upper bound estimation, if the number of solutions associated with a variable value is inferred to be zero, then we can safely remove this value from the corresponding domain at compile time. Moreover, recall that the 1-expansion n-cliques algorithm is applied to every variable to obtain the number of solutions for all the variable values. Knowing that a value cannot lead to a solution, we can also safely remove the corresponding vertex in the consistency graph without affecting correctness. This reduction can indeed improve both the efficiency and precision of the 1-expansion n-cliques algorithm.

Therefore, for the example problem, the predicate `generator`/2 can be specialized into the following new generator predicates at compile time:

```
gen_a(1).      gen_a(2).
```

Benchmarks	SF	UGO	RGO	RGO/UGO	RGO/SF
send-more-money	34.6400	0.0514	0.0471	0.9163	0.0013
five-houses	0.3451	0.0856	0.0297	0.3469	0.0860
job-scheduling	4.2800	0.4187	0.0511	0.1220	0.0119
liar-puzzle	0.0067	0.0065	0.0060	0.9230	0.8955
network-flow	0.7855	0.4531	0.4418	0.9750	0.5624

Benchmark	SF	ULO	RLO	RLO/ULO	RLO/SF
send-more-money	34.6400	0.0542	0.0482	0.8892	0.0013
five-houses	0.3451	0.03607	0.0312	0.8666	0.0904
job-scheduling	4.2800	0.3138	0.0378	0.1204	0.0088
liar-puzzle	0.0067	0.0062	0.0060	0.9677	0.8955
network-flow	0.7855	0.4531	0.4418	0.9750	0.5624

Table 2: Measurements for domain reducing optimization (times in seconds)

```
gen_b(3).      gen_b(4).      gen_b(8).      gen_b(9).
gen_c(3).      gen_c(4).      gen_c(6).      gen_c(7).
gen_d(3).      gen_d(4).      gen_d(5).
gen_e(11).     gen_e(12).
gen_f(9).      gen_f(10).     gen_f(11).
gen_g(13).     gen_g(14).
gen_end(14).   gen_end(15).
```

Experimental measurements for domain reducing optimization have been conducted on the benchmark programs. Among them, the eight-queens program, the magic squares program and the map coloring program have no variable values that can be removed. Since the program transformation from a generator using recursive predicate to a generator using a set of facts usually also reduce the execution time, we use the version using facts to measure the sole effects from the domain reducing optimization. For each goal ordering technique, the programs are executed for a version without domains reduced (UGO and ULO) and a version with domains reduced (RGO and RLO). The result is shown in the Table 2. From the measurements, combining the goal ordering transformation and the domain reducing optimization can speed up the execution of the send-more-money puzzle program and the job scheduling program by a factor of more than a hundred. Notice also that for the five-houses puzzle program and the liar

puzzle program, the 1-expansion n-cliques algorithm infers that for each variable at most one value in the domain can satisfy the constraints in the problem. Consequently, after reducing the variable domains, these programs can be solved without any backtracking at runtime.

In the benchmark programs above, the goal ordering transformation is applied ahead of the domain reducing optimization. It is reasonable to reverse this application order. But, since domains will be represented as sets of atoms, instead of integer intervals, after applying the domain reducing optimization, we will be forced to use set manipulations, rather than integer interval arithmetic, for arithmetic constraints when applying the goal ordering transformation. We use the original order because set manipulations are usually much more expensive than integer interval arithmetic. However, this may cause some loss in precision.

3.3 Determining Instantiation Order of Variable Values

For some applications, such as theorem proving, planning and vision problems, finding a single solution is sufficient. For some other applications, such as combinatorial optimization problems, an optimal solution is sought. In this latter situation, usually, the branch and bound algorithm is employed, and we will try to find a first solution as soon as possible so that we can start the pruning early. In all these cases, the order in which the domain values are instantiated to the variables may have profound effect on the program performance.

In the example problem, suppose we are interested in obtaining any of the schedules that satisfy the schedule constraints. Then we can use the number of solutions information to determine the instantiation order of variable values. A simple heuristic is to choose the value that is associated with the most number of solutions as the first value to be instantiated. Thus using the information about the number of solutions, we can rearrange the order of the clauses in the generator predicates as follows:

```
gen_a(1).     gen_a(2).
gen_b(3).     gen_b(8).     gen_b(9).     gen_b(4).
gen_c(3).     gen_c(6).     gen_c(7).     gen_c(4).
gen_d(3).     gen_d(4).     gen_d(5).
gen_e(12).    gen_e(11).
gen_f(11).    gen_f(10).    gen_f(9).
gen_g(14).    gen_g(13).
gen_end(15).  gen_end(14).
```

Benchmarks	GNO	GCO	GCO/GNO	LNO	LCO	LCO/LNO
send-more-money	22.370	9.540	0.873	22.570	19.611	0.868
eight-queens	41.590	13.530	0.325	32.429	11.201	0.345
job-scheduling	0.292	0.243	0.832	0.280	0.241	0.860
magic-squares	19.251	5.089	0.472	50.240	0.790	0.015
network-flow	0.730	0.289	0.395	0.730	0.289	0.395

Table 3: Measurements for clause ordering transformation (times in milliseconds)

Experimental measurements for this clause ordering transformation have been conducted on the benchmark programs. Among them, the 1-expansion n-cliques algorithm infers that for the five-houses puzzle program and the liar puzzle program, there is at most one variable value satisfying the constraints in the program for each variable, and for the map coloring program, all the variable values are associated with the same number of solutions for each variable. Thus the measurements for these three programs are not included. For each goal ordering technique, the programs are executed for a version using natural domain value order (GNO and LNO) and a version using the order generated by our clause ordering technique (GCO and LCO). The result is shown in the Table 3. The result shows that for all the benchmark programs, the version using clause ordering technique performs better than the version using natural order.

3.4 Managing Parallelism

In some cases, we may want to collect all the solutions satisfying the constraints in the program and to perform some post-processing upon them. For instance, in the example problem, suppose we want to compute both the latest starting time and the earliest completion time among the legal schedules. Then the following predicate `schedules/3` specifies the desired computation.

```
schedules(S,MaxS,MinC) :-
    setof([SA,SB,SC,SD,SE,SF,SG,SEnd],
        schedule(1-15,SA,SB,SC,SD,SE,SF,SG,SEnd),S),
    latest_start(S,MaxS), earliest_completion(S,MinC).
```

It first uses predicate `setof/3` to collect all the legal schedules in the duration `[1..15]` as a list `S`, then it computes the latest starting time and the earliest completion time among the

list of legal schedules separately.

In parallel systems ROLOG [17] and &-Prolog [16], independent subgoals are typically executed in parallel. Since literals `latest_start`/2 and `earliest_completion`/2 are independent of each other, they will be executed in parallel in these systems. However, parallel execution of these literals benefits only when the cost saved from the parallel execution outweights the cost spent for managing the parallelism [8]. The performance of parallel systems usually begins to degrade when the systems are choked with a lot of small grain processes. Thus, suitable process granularity control such that "sufficiently" small processes are executed sequentially may often improve the performance of parallel systems. However, the information about the costs of the literals is required in order to perform suitable process granularity control. Because the cost of executing the two literals in the example above depends on the length of the list of legal schedules, the decision on whether to execute them in parallel or in serial depends on the number of solutions generated by literal `schedule`/9. Knowing that `schedule`/9 will generate 66 solutions, we can plan to execute them in parallel. On the other hand, if the given duration is `[1..14]` instead of `[1..15]`, then only 6 solutions are generated; in this case, we might choose to execute them sequentially.

In distributed systems, the communication cost among processors plays a more crucial role than in shared-memory systems. In these systems, appropriate process granularity control should also take the communication cost into account [19]. In general, the communication cost depends on the size of the input and output arguments in the predicates. For the example problem, if literal `earliest_completion`/2 is spawned as a new process to a remote processor, the communication cost would be a function in terms of the number of legal schedules.

Apart from process granularity control, the process computation cost and communication cost can also be used to schedule the process migration among processors [35]. In the systems using on-demand scheduling method, when a processor becomes idle, it will inquire and request tasks from other busy processors. In these systems, the cost information is used by the busy processors to guide the choosing of tasks for migration. In general, the task that has the largest computation cost and the smallest communication cost is chosen to migrate to a remote processor.

4 Conclusions

Combinatorial problems are often specified declaratively in logic programs with constraints over finite domains. This paper has presented several program transformation and optimiza-

tion techniques for improving the performance of the class of program with finite-domain constraints. These techniques allow automatic transformation of declaratively specified programs into more efficient programs. The techniques are based on the information about the number of solutions of constraint satisfaction problems. The techniques include planning the evaluation order of body goals, reducing the domain of variables, planning the instantiation order of variable values, and controlling process granularity and scheduling processes among processors in parallel or distributed systems.

Since the decision versions of many constraint satisfaction problems are NP-complete, if P $\neq$ NP, there is no polynomial time algorithm for computing the number of solutions for constraint satisfaction problems. This paper has presented a simple greedy algorithm for computing an upper bound on the number of solutions for constraint satisfaction problems over finite domains. The time complexity of this algorithm is $O(v(v+e))$ for a problem whose corresponding consistency graph contains v vertices and e edges. Based on this simple algorithm, a set of flexible polynomial time algorithms has also been developed. A user can choose the appropriate algorithm according to specific efficiency and precision requirements.

Acknowledgements

The author would like to thank Saumya Debray for many valuable comments on the material of this paper.

References

[1] M. Bartusch, *Optimierung von Netzplaenen mit Anordnungsbezziehungen bei Knappen Betriebsmitteln*, PhD thesis, Fakultaet fur Mathematik und Informatik, Universitaet Passau, F.R.G., 1983.

[2] M. Bruynooghe and L. M. Pereira, "Deduction Revision by Intelligent Backtracking," *Implementation of PROLOG*, edited by J. A. Campbell, Ellis Horwood, 1984, pp. 194–215.

[3] M. Carlsson and J. Widen, *SICStus Prolog User's Manual*, Swedish Institute of Computer Science, Kista, Sweden, 1988.

[4] K. L. Clark and F. G. McCabe, "The Control Facilities of IC-Prolog," *Expert Systems in Microelectronic Age*, edited by D. Michie, University of Edinburgh, Scotland, 1979, pp. 153–167.

[5] A. Colmerauer, H. Kanoui, and M. Van Caneghem, "Prolog, Bases Theoriques et Developpements Actuels," *Techniques et Sciences Informatiques*, 2 (4), 1983, pp. 271–311.

[6] S. A. Cook, "The Complexity of Theorem-Proving Procedures," *Conference Record 3rd Annual ACM Symposium on Theory of Computing*, 1971, pp. 151–158.

[7] D. De Schreye, D. Pollet, J. Ronsyn and M. Bruynooghe, "Implementing Finite-domain Constraint Logic Programming on Top of a PROLOG-system with Delay-mechanism," Report CW-104, Department of Computer Science, K. U. Leuven, Leuven, Belgium, December 1989.

[8] S. K. Debray, N. Lin and M. Hermenegildo, "Task Granularity Analysis in Logic Programs," *Proceedings of ACM SIGPLAN'90 Conference on Programming Language Design and Implementation*, June 1990, pp. 174–188.

[9] S. K. Debray and N. Lin, "Static Estimation of Query Sizes in Horn Programs," *Proceedings of Third International Conference on Database Theory*, Paris, France, December 1990, pp. 514–528.

[10] S. K. Debray and N. Lin, "Cost Analysis of Logic Programs," to appear in *ACM Transactions on Programming Languages and Systems.*

[11] R. Dechter and J. Pearl, "Network-Based Heuristics for Constraint-Satisfaction Problems," *Artificial Intelligence* 34, (1988), pp. 1–38.

[12] E. C. Freuder, "Synthesizing constraint expressions," *Communications of the ACM* 21, 11 (November 1978), pp. 958–966.

[13] J. Gaschnig, *Performance measurement and analysis of certain search algorithms*, Ph.D. thesis, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA, 1979.

[14] R. M. Haralick and G. L. Elliot, "Increasing Tree Search Efficiency for Constraint Satisfaction Problems," *Artificial Intelligence* 14, (1980), pp. 263–313.

[15] N. Heintze and J. Jaffar, "A Finite Presentation Theorem for Approximating Logic Programs," *Proceedings of ACM Symposium Principles of Programming Languages*, San Francisco, California, Jan. 1990, pp. 197–209.

[16] M. V. Hermenegildo, *An Abstract Machine Based Execution Model for Computer Architecture Design and Efficient Implementation of Logic Programs in Parallel*, PhD thesis, The University of Texas at Austin, Austin, 1986.

[17] L. V. Kalé, *Parallel Architectures for Problem Solving*, PhD thesis, SUNY, Stony Brook, 1985.

[18] R. M. Karp, "Reducibility Among Combinatorial Problems," *Complexity of Computer Computations*, edited by R. E. Miller and J. W. Thatcher, Plenum Press, New York, 1972, pp. 85–103.

[19] A. King and P. Soper, "Granularity Control for Concurrent Logic Programs," *Fifth International Symposium on Computer and Information Sciences*, Nevsehir, Turkey, October 1990.

[20] V. Kumar and Y.-J. Lin, "A Data-Dependency-Based Intelligent Backtracking Scheme for PROLOG," *Journal of Logic Programming*, (1988), pp. 165–181.

[21] J. L. Lauriere, "A Language and a Program for Stating and Solving Combinatorial Problems," *Artificial Intelligence*, 10 (1), 1978, pp. 29–127.

[22] A. K. Mackworth, "Consistency in Networks of Relations," *Artificial Intelligence* 8, (1977), pp. 99–118.

[23] A. K. Mackworth and E. C. Freuder, "The Complexity of Some Polynomial Network Consistency Algorithms for Constraint Satisfaction Problems," *Artificial Intelligence* 25, (1985), pp. 65–74.

[24] P. Mishra, "Toward a Theory of Types in Prolog", *Proceedings of 1984 IEEE Symposium on Logic Programming*, Atlantic City, 1984, pp. 289-298

[25] R. Mohr and T. C. Henderson, "Arc and Path Consistency Revisited," *Artificial Intelligence* 28, (1986), pp. 225–233.

[26] U. Montanari, "Networks of Constraints: Fundamental Properties and Applications to Picture Processing," *Information Sciences* 7, (1974), pp. 95–132.

[27] L. Naish, *Negation and Control in Prolog*, Lecture Notes in Computer Science 238, Springer-Verlag, Germany, 1986.

[28] K. S. Natarajan, "On Optimizing Backtrack Search for All Solutions to Conjunctive Problems," Research Report RC 11982, IBM T. J. Watson Research Center, Yorktown Heights, New York, 1986.

[29] B. Nudel, "Consistent-Labeling Problems and their Algorithms: Expected-Complexities and Theory-Based Heuristics," *Artificial Intelligence* 21, (1983), pp. 135–178.

[30] C. Pyo and U. S. Reddy, "Inference of Polymorphic Types for Logic Programs," *Proceedings of North American Conference on Logic Programming*, Cleveland, OH, 1989, pp. 1115–1134.

[31] I. Rivin and R. Zabih, "An Algebraic Approach to Constraint Satisfaction Problems," *Proceedings of Eleventh International Joint Conference on Artificial Intelligence*, August, 1989, pp. 284–289.

[32] H. Seki and K. Furukawa, "Notes on Transformation Techniques for Generate and Test Logic Programs," *Proceedings of IEEE Symposium on Logic Programming*, 1987, pp. 215–223.

[33] D. E. Smith and M. R. Genesereth, "Ordering Conjunctive Queries," *Artificial Intelligence* 26 (1985), pp. 171–215.

[34] R. E. Tarjan, *Data Structures and Network Algorithms*, Society for Industrial and Applied Mathematics, Philadelphia, Pennsylvania, 1983.

[35] E. Tick, "Compile-Time Granularity Analysis for Parallel Logic Programming Languages," *New Generation Computing* 7, (1990), pp. 325–337.

[36] P. Van Hentenryck, *Constraint Satisfaction in Logic Programming*, The MIT Press, Cambridge, Massachusetts, 1989.

[37] D. Waltz, "Understanding line drawings of scenes with shadows," *The Psychology of Computer Vision*, edited by P. H. Winston, McGraw-Hill, New York, 1975.

[38] D. H. D. Warren, "Efficient Processing of Interactive Relational Database Queries Expressed in Logic," *Proceedings of Seventh International Conference on Very Large Data Bases*, 1981, pp. 272–281.

[39] E. Yardeni and E. Shapiro, "A Type System for Logic Programs," *Concurrent Prolog: Collected Papers*, vol. 2, edited by E. Shapiro, pp. 211-244.