Estimating the Number of Solutions for Constraint Satisfaction Problems *

Nai-Wei Lin
Department of Computer Science
The University of Arizona
Tucson, AZ 85721
naiwei@cs.arizona.edu

March 1992

Abstract

Knowledge about the number of solutions of constraint satisfaction problems can be used to improve the efficiency of logic programs and deductive databases, e.g., to control task granularity in parallel systems, to map predicates to processors in distributed systems, or to plan the evaluation order of body goals and the instantiation order of variable values. Many constraint satisfaction problems are NP-complete. Thus it is very unlikely to find an efficient algorithm to compute the number of solutions for constraint satisfaction problems. This paper presents a greedy approximation algorithm for estimating the number of solutions for constraint satisfaction problems on finite domain. The time complexity of this algorithm is $O(n^3m^3)$ for a problem involving n variables and m domain values. Based on this algorithm, a set of flexible algorithms is also developed. The user can choose the appropriate algorithm according to the efficiency and precision requirements.

1 Introduction

Many problems arising in artificial intelligence and operations research can be formulated as a constraint satisfaction problem (CSP). A CSP involves a set of n variables, a domain of values, and a set of constraints on the variables. A solution of a CSP is an n-tuple of assignments of domain values to variables such that all the constraints in the problem are satisfied. Familiar problems which can be formulated as a CSP include *boolean satisfiability*: deciding whether a boolean formula is satisfiable; *graph coloring*: coloring a graph so that adjacent vertices have different colors; *subgraph isomorphism*:

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

given two graphs, deciding whether one graph is isomorphic to a subgraph of the other one, and so on. For a survey on the algorithms for solving CSPs, the reader is referred to [18].

The motivation for this work is to infer useful information to support various optimization techniques for improving the efficiency of logic programs and deductive databases. Since CSPs appear frequently in logic programs and deductive databases, knowledge about the number of solutions of CSPs can be applied in a variety of ways. For example, the efficiency of parallel logic programs can be improved by suitable process granularity control [4]. Because of nondeterminism, to compute the granularity of a process, we need to know the number of solutions generated by each procedure [6]. In distributed environment, the performance of a system is affected by the amount of communication in the system, which usually corresponds to the number of solutions generated by each distributed task. The number of solutions information can be used to properly map tasks to processors. The efficiency of query evaluation in deductive databases depends on the evaluation order of subgoals [21, 24]. Knowledge about the number of solutions generated by each subgoal can be applied to plan the appropriate evaluation order among subgoals. For systems where the instantiation order of variable values can be controlled, the estimates for the number of solutions generated for each value instantiation can be employed to determine the instantiation order of values so that the first solution can be efficiently produced [7].

Many CSPs which decide whether there exists a solution satisfying all the constraints in the problem are NP-complete, such as boolean satisfiability, graph k-colorability and subgraph isomorphism [3, 12]. Thus it is very unlikely to find an efficient algorithm to compute the number of solutions for CSPs. Rivin and Zabih have proposed an algorithm for computing the number of solutions for CSPs by transforming a CSP into an integer linear programming problem [20]. If the CSP 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 approximation algorithms for estimating the number of solutions for CSPs on finite domain of discrete values. This paper presents an $O(n^3m^3)$ approximation algorithm for a problem involving n variables and m domain values based on the greedy method. The basic idea of this algorithm is recursively reducing a CSP involving n variables to a CSP involving $n-1$ variables. Through the reduction, the number of solutions of the CSP involving $n-1$ variables is made to be an upper bound on the number of solutions of the CSP involving n variables. Based on this algorithm, a set of flexible polynomial time approximation algorithms is also developed. The user can choose the appropriate algorithm according to the efficiency and precision requirements. These algorithms are particularly useful for problems where constraints are explicitly expressed as a set of arithmetic constraints. It is very difficult to infer nontrivial number of solutions information for these problems without considering the net effects of the set of arithmetic constraints. Estimation of the number of solutions for this class of problems has not been addressed in previous work [5, 21, 24].

We will use n-queens and inverse n-queens problems as our running examples. The n-queens problem requires the placement of n 'queens' on an $n \times n$ chessboard such that each queen is placed in a different row and no pair of queens are in the same column, or diagonals. The n-queens problem can be formulated

as a CSP as follows: let x_i denote the queen placed in i^{th} row and the domain of values $1, \ldots, n$ be the column indices. Then the constraints are $x_j \neq x_i$ and $x_j \neq x_i \pm (j-i)$, for $1 \leq i < j \leq n$. The inverse n-queens problem, on the other hand, requires the placement of queens such that every pair of queens are in the same column, or diagonals [18]. Using the same notations as the n-queens problem, the constraints for inverse n-queens problem are $x_j = x_i$ or $x_j = x_i \pm (j-i)$, for $1 \leq i < j \leq n$.

2 Number of Solutions and Number of N-cliques

In this paper we consider the following class of **binary CSPs**: a 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.[1] A solution of a CSP is an n-tuple of assignments of domain values to variables such that all the constraints in the problem are satisfied.

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, $x_i \leftarrow d_j$, 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 of 3-queens problem 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 of a CSP p involving n variables is an n-tuple of assignments of domain values to variables such that all the constraints are satisfied, every pair of assignments in s is consistent. Thus 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. We will use $K(G, n)$ to denote the number of n-cliques in a consistency graph G. Since there is no 3-clique in the consistency graph of 3-queens problem, as shown in Figure 1, there is no solution for 3-queens problem.

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. The **adjacency graph** of v, $Adj_G(v)$, 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 $V =$*

[1] refer to [17] for relationship with n-ary constraints.

$\{v_1, \ldots, v_m\}$,

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

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 $K(G,n)$.

3 An Upper Bound on 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 intention 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 $K(H,n) = min_{e \in E_H}\{W_H(e)\}$, and $K(G,n) = \sum_{H \in S} K(H,n)$. The intuition behind this formulation will shortly become clear. Let the weighted consistency graph corresponding to a consistency graph $G = (V,E)$ be $G' = (V,E,W)$, where $W(e) = 1$, for all $e \in E$. We then have $K(G,n) = K(G',n)$ by definition, and can concentrate on weighted consistency graph from now on.

We 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 edge $\langle ij, gh \rangle$ joins the vertices denoting $x_i \leftarrow d_j$ 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 3 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 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 3.1 *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 $min_{e\in E_H}\{W_H(e)\} \geq min_{e\in E_H}\{W_1(e)\} + min_{e\in E_H}\{W_2(e)\}$. If H is in either G_1 or G_2, but not both, then $min_{e\in E_H}\{W_H(e)\} \geq min_{e\in E_H}\{W_1(e)\}$ and $min_{e\in E_H}\{W_2(e)\} = 0$, or $min_{e\in E_H}\{W_H(e)\} \geq min_{e\in E_H}\{W_2(e)\}$ and $min_{e\in E_H}\{W_1(e)\} = 0$. If H is in neither G_1 nor G_2, then $min_{e\in E_H}\{W_H(e)\} > 0$, and $min_{e\in E_H}\{W_1(e)\} = min_{e\in E_H}\{W_2(e)\} = 0$. □

Theorem 3.1 verifies that graph addition is a way to combine the set of adjacency graphs such that the upper bound condition holds. 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 adjacent graph of v in 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 3.2 *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 pair of 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 $N_G(v_i)$, $min_{e\in E_{G_i}}\{W_{G_i}(e)\} = min_{e\in E_{A_i}}\{W_{A_i}(e)\}$. Therefore, $K(G_i, n) = K(Adj_G(v_i), n-1)$. □

Theorem 3.3 *Let G be a weighted consistency graph of order (n,m). Then for each assignment set $V = \{v_1, \ldots, v_m\}$,*

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

Proof By Theorems 3.1 and 3.2. □

4 An Approximation Algorithm

We are now ready to present a simple greedy approximation 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 3.3 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 bipartite weighted 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 n-cliques algorithm. Each adjacency graph can be constructed in time $O(n^2m^2)$; and each graph addition can be performed in time $O(n^2m^2)$. Therefore, the total time required for Line 4 is $O(n^2m^3)$. Taking the loop into account, Lines 3-5 require time $O(n^3m^3)$. 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 $O(n^3m^3)$.

We have applied n-cliques algorithm on n-queens and inverse n-queens problems. The results are shown in Figures 3 and 4. The results show that the estimates for the inverse n-queens problem are very good, apart from 4-queens, all the estimates are exact; on the other hand, the estimation error for n-queens problem grows quickly as the graph density increases. The reason for the difference is that the graph density and the number of solutions for inverse n-queens problem are linearly increasing, while the graph density and the number of solutions for n-queens problem are exponentially increasing. In practice, we can anticipate very few problems will generate as ample solutions as the n-queens problem does.

5 Improving Estimation Accuracy

In this section we investigate three methods for improving the estimation accuracy of n-cliques algorithm: (1) **network consistency**; (2) **exact expansion**; and (3) **assignment memorization**. For

all three methods, full application will lead to exact estimation and exponential time. Therefore, we will examine the tradeoff between efficiency and accuracy of these methods.

5.1 Network Consistency

CSPs are usually solved by backtracking algorithm. Since basic backtracking algorithm may incur significant inefficiency [2], a class of network consistency algorithms has been proposed to improve the efficiency of backtracking algorithm [8, 9, 10, 13, 16, 17, 23]. The basic idea behind the network consistency algorithms is as follows. Each constraint in the problem only makes the local consistency between assignments explicit. Through exploiting some global consistency (i.e., consistency involving more than two variables) between assignments, we may remove beforehand some of the domain values from testing at every stage of the backtracking algorithm.

We can use the same idea to reduce the consistency graph by removing the edges which are unable to satisfy global consistency. 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 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 n-cliques algorithm, we can just replace the formula $\bigoplus_{j=1}^{m} Adj_G(v_{i,j})$ in 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 which achieve 2-consistency and 3-consistency are $O(n^2m^3)$ and $O(n^3m^5)$ respectively [14].

5.2 Exact Expansion

Exact expansion method tries to balance the accuracy of Formula (3) and the efficiency of Formula (4). The intention 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 $O(m)$.

5.3 Assignment Memorization

In 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 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, the 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 such that the graph addition may take advantage of that information.

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 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.

We have incorporated the network consistency, exact expansion and assignment memorization algorithms into the n-cliques algorithm and applied it on n-queens problem. The result is shown in Figure 5. The 3-consistency algorithm achieves 3-consistency of a graph, the 2-expansion algorithm expands the Formula (3) twice, and the 2-memorization algorithm memorizes the most recent two variable assignments, respectively. The result shows that lower order network consistency algorithms only have effect on sparse graphs. That is because the graph is more likely to be consistent when the graph density increases. However, for applications where network consistency algorithms work well for finding solutions, this method for estimating the number of solution will also work well. On the other hand, the exact expansion algorithms and the assignment memorization algorithms can still significantly improve the estimation accuracy as the graph density increases. Moreover, the estimation error growth rate for the assignment memorization algorithms is slower than for the exact expansion algorithms. The reason is that the assignment memorization algorithms improve the accuracy evenly at every iteration of the process, while the exact expansion algorithms improve the accuracy mainly at the beginning iterations of the process. Because these methods can achieve any degree of accuracy by paying the price on efficiency, it is also possible to construct adaptive algorithms which will adaptively decide the amount of efforts to invest based on the density of the graphs and the accuracy requirements.

6 Applications

The algorithms introduced in this paper are particularly useful for problems where constraints are explicitly expressed as a set of arithmetic constraints. It is very difficult to infer nontrivial number of solutions information for these problems without considering the net effects of the set of arithmetic constraints. Estimation of the number of solutions for this class of problems has not been addressed in previous work [5, 21, 24]. Given the domain information about the variables in the arithmetic constraints,[2] the consistency graph corresponding to the constraints can be efficiently built, and nontrivial number of solutions information can be efficiently inferred. This section gives an example to illustrate the applications of the number of solutions information. The example problem is to find the

[2]Domain information can be inferred at compile time [11, 15, 19, 25] or declared by user [22].

legal schedules for a project that satisfies a variety of constraints among tasks as well as to compute the latest start time and the earliest completion time among the legal schedules [1, 22]. The constraints may include *distance constraints* which specify the delay required between the start of two tasks due to sharing of resources, *precedence constraints* which specify the precedence among the tasks, and several other constraints. An example of distance constraints is shown in Table 1 and specified in the following predicate:

$$
\begin{array}{l}
distance\_constraints(SB, SC, SD) \ :- \\
\quad SC \geq SB + 1, \qquad SD \geq SC + 1.
\end{array}
$$

The variables SB, SC and SD in the predicate denote the start time for the tasks A, B and C respectively. An example of precedence constraints is shown in Table 2 and specified in the following predicate:

$$
\begin{array}{l}
precedence\_constraints(SA, SB, SC, SD, SE, SF, SEnd) \ :- \\
\quad SB \geq SA + 1, \qquad SC \geq SA + 1, \qquad SD \geq SA + 1, \qquad SE \geq SB + 5, \\
\quad SE \geq SC + 3, \qquad SF \geq SD + 5, \qquad SF \geq SE + 2, \qquad SEnd \geq SF + 1.
\end{array}
$$

A legal schedule which satisfies the above distance and precedence constraints in a given duration can be specified as follows:

$$
\begin{array}{l}
schedule\_constraints(Duration, SA, SB, SC, SD, SE, SF, SEnd) \ :- \\
\quad generator(Duration, SA, SB, SC, SD, SE, SF, SEnd), \\
\quad distance\_constraints(SA, SB, SC, SD, SE, SF, SEnd), \\
\quad precedence\_constraints(SA, SB, SC, SD, SE, SF, SEnd).
\end{array}
$$

The predicate $generator/8$ generates all the possible time slots in the given duration for the variables. The following predicate $schedule/3$ uses predicate $bagof/3$ to gather all the legal schedules in the duration $0 \sim 10$, then compute the latest start time and earliest completion time among the legal schedules.

$$
\begin{array}{l}
schedule(S, MaxS, MinC) \ :- \\
\quad bagof([SA, SB, SC, SD, SE, SF, SEnd], \\
\qquad schedule\_constraints(10, SA, SB, SC, SD, SE, SF, SEnd), S), \\
\quad latest\_start(S, MaxS), \\
\quad earliest\_completion(S, MinC).
\end{array}
$$

The predicate $schedule/3$ can produce 10 legal schedules, and the inequality predicate $\geq/2$ in predicates $distance\_constraints/3$ and $precedence\_constraints/7$ will be executed a total of 32,236,567 times. Using the domain information about the duration $0 \sim 10$ for the variables $SA, SB, \ldots, SEnd$, we can apply the n-cliques algorithm to estimate the number of solutions for each of the inequalities in predicate

precedence_constraints/7. Since they all are binary constraints, we can get the exact estimate for each of them. Based on the number of solutions information, we can rearrange the execution order among the inequalities so that the most likely failed inequalities will be executed first:

$$\begin{aligned} &precedence\_constraints(SA, SB, SC, SD, SE, SF, SEnd) \ :- \\ &\quad SE \geq SB + 5, \qquad SF \geq SD + 5, \qquad SE \geq SC + 3, \qquad SF \geq SE + 2, \\ &\quad SB \geq SA + 1, \qquad SC \geq SA + 1, \qquad SC \geq SA + 1, \qquad SEnd \geq SF + 1. \end{aligned}$$

Using this ordering, the predicate $\geq/2$ is executed a total of 31,703,353 times. We can use the n-cliques algorithm to further estimate the net effects of the set of inequalities for predicates *distance_constraints*/3 and *precedence_constraints*/7. The estimated number of solutions for *distance_constraints*/3 is 165, and it is 71 for *precedence_constraints*/7, both are exact estimates. These information enables us to reverse the execution order of the literals *distance_constraints*/3 and *precedence_constraints*/7 in predicate *schedule_constraints*/8. Under this ordering, the predicate $\geq/2$ is executed a total of 23,863,814 times, a significant reduction from the original program.

If we are only interested in getting one solution, we can use the number of solutions information produced for each of the variable values to plan the instantiation order of variable values. For example, for variable SE, it will greatly reduce the execution time to try values 6 or 7 first instead of starting with value 0. In parallel or distributed systems, the number of solutions information for each inequality is needed to estimate the granularity and communication cost of predicate *schedule_constraints*/8; and the number of solutions information for predicate *schedule_constraints*/8, which is 10 and can be estimated exactly by n-cliques algorithm, is needed to estimate the granularity and communication cost of predicates *latest_start*/2 and *earliest_completion*/2. These information can facilitate the management of task granularity and task distribution.

7 Conclusions

Knowledge about the number of solutions of constraint satisfaction problems can be used to improve the efficiency of logic programs and deductive databases, e.g., to control task granularity in parallel systems, to map predicates to processors in distributed systems, or to plan the evaluation order of body goals and the instantiation order of variable values. Many constraint satisfaction problems are NP-complete. Thus it is very unlikely to find an efficient algorithm to compute the number of solutions of constraint satisfaction problems. This paper has presented a greedy approximation algorithm for estimating the number of solutions for constraint satisfaction problems on finite domain. The time complexity of this algorithm is $O(n^3m^3)$ for a problem involving n variables and m domain values. Based on this algorithm, a set of flexible algorithms has also been developed. The user can choose the appropriate algorithm according to the efficiency and precision requirements.

Acknowledgements

The author likes 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] D. G. Bobrow and B. Raphael, "New Programming Languages for AI Research," *Computing Survey* 6, (1974), pp. 153–174.

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

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

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

[6] S. K. Debray and N. Lin, "Automatic Complexity Analysis of Logic Programs," *Proc. Eighth International Conference on Logic Programming*, Paris, June, 1991, pp. 599–613.

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

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

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

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

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

[12] 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.

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

[14] 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.

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

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

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

[18] B. A. Nadel, "Constraint Satisfaction Algorithms," *Computational Intelligence* 5, (1989), pp. 188–224.

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

[20] I. Rivin and R. Zabih, "An Algebraic Approach to Constraint Satisfaction Problems," *Proc. Eleventh IJCAI*, August, 1989, pp. 284–289.

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

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

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

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

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

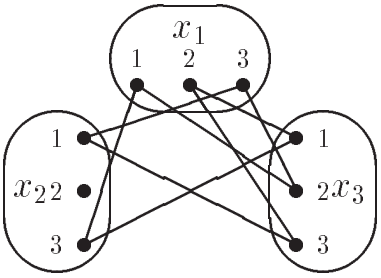
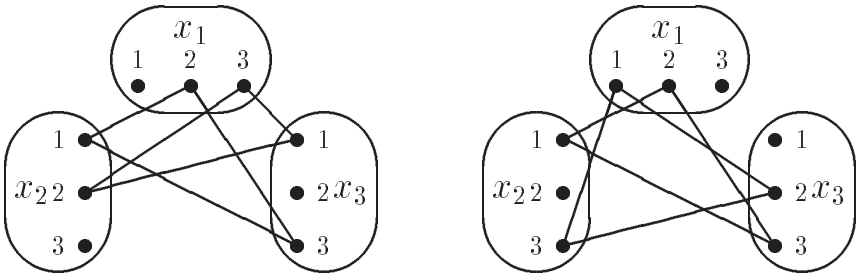
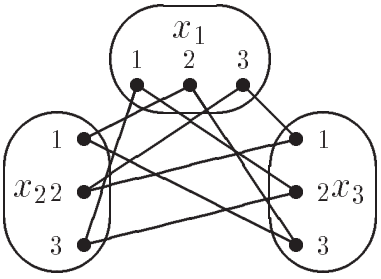


Figure 1: The consistency graph of 3-queens problem



G_1 G_2



$G_1 \oplus G_2$

Figure 2: An example of graph addition

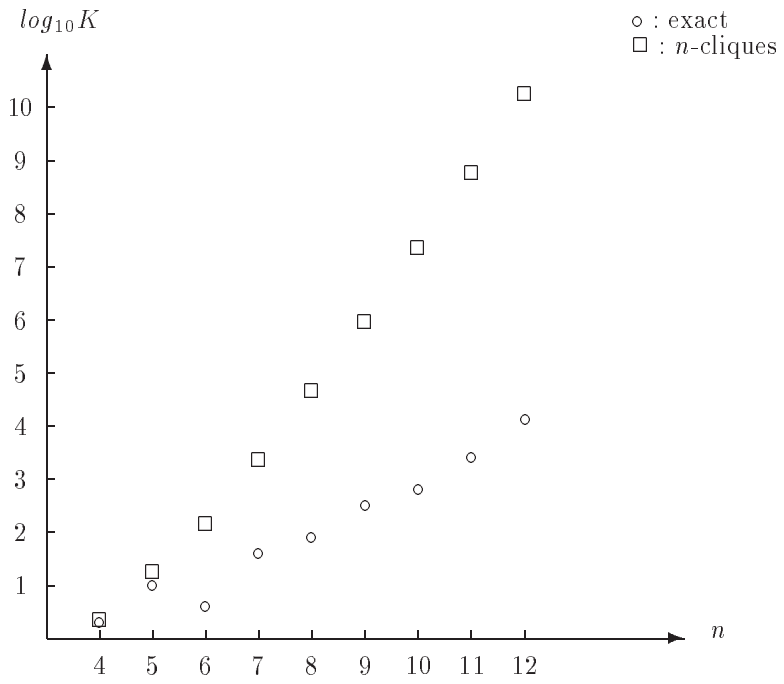


Figure 3: The approximate number of solutions of n-queens problem computed by n-cliques algorithm

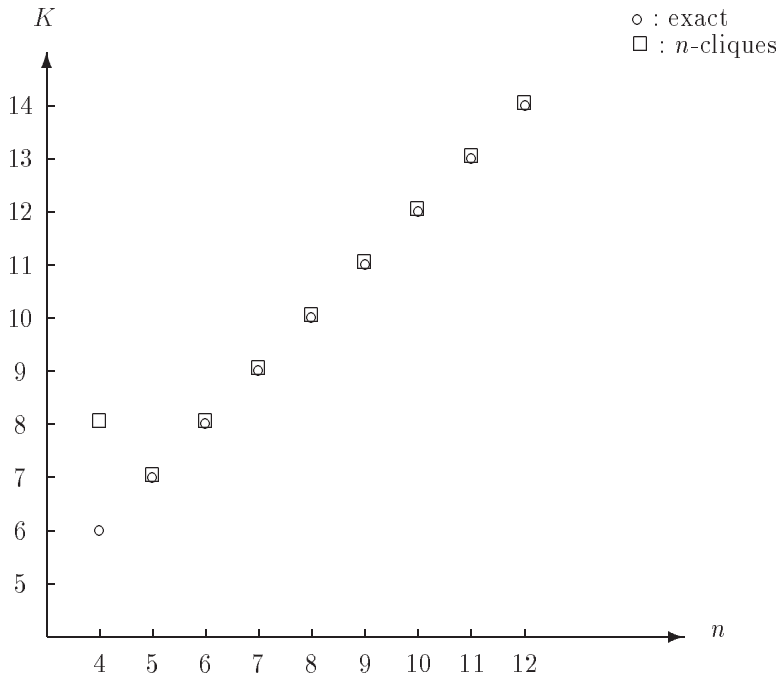


Figure 4: The approximate number of solutions of inverse n-queens problem computed by n-cliques algorithm

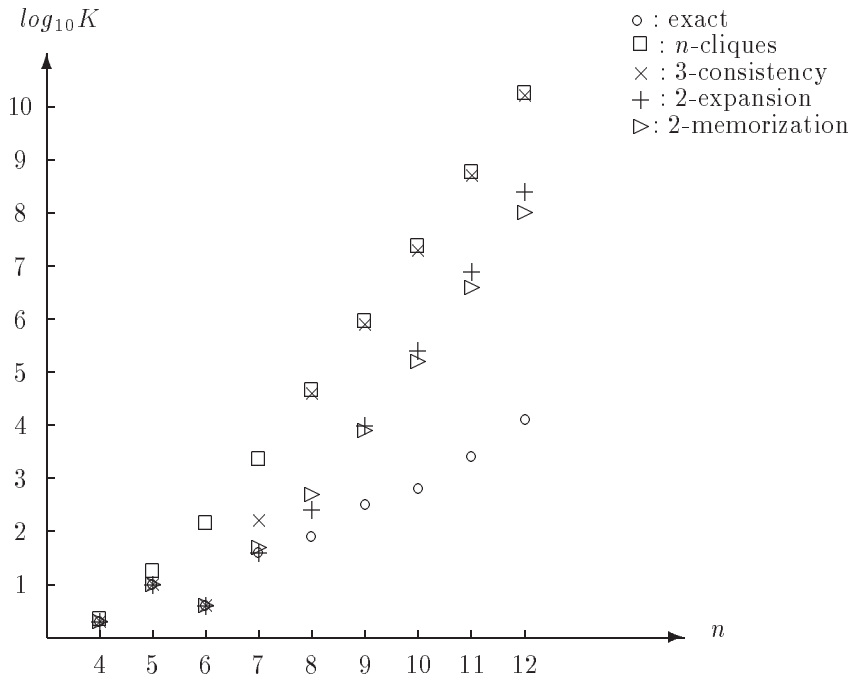


Figure 5: The approximate number of solutions of n-queens problem computed by n-cliques algorithm incorporated with the network consistency, exact expansion and assignment memorization algorithms

task	delay	previous task
C	1	B
D	1	C

Table 1: An example of distance constraints

task	duration	previous tasks
A	1	-
B	5	A
C	3	A
D	5	A
E	2	B,C
F	1	D,E

Table 2: An example of precedence constraints