Efficient Algorithms for Functional Constraints*

Yuanlin Zhang[1], Roland H.C. Yap[2],
Chendong Li[1], and Satyanarayana Marisetti[1]

[1] Texas Tech University, USA
[2] National University of Singapore, Singapore

Abstract. *Functional constraints* are an important constraint class in Constraint Programming (CP) systems, in particular for Constraint Logic Programming (CLP) systems. CP systems with finite domain constraints usually employ CSP-based solvers which use local consistency, e.g. arc consistency. We introduce a new approach which is based instead on *variable substitution*. We obtain efficient algorithms for reducing systems involving functional and *bi-functional constraints* together with other non-functional constraints. It also solves globally any CSP where there exists a variable such that any other variable is reachable from it through a sequence of functional constraints. Our experiments show that variable elimination can significantly improve the efficiency of solving problems with functional constraints.

1 Introduction

Functional constraints are a common class of constraints occurring in Constraint Satisfaction Problem(s) (CSP) [10, 11, 7]. In Constraint Programming (CP) systems such as Constraint Logic Programming (CLP), functional constraints also naturally arise as primitive constraints and from unification. Finite domain is a widely used and successful constraint domain for CLP. In this context, functional constraints (e.g., those in CHIP [11]), as primitive constraints, can facilitate the development of more efficient constraint solvers underlying CLP systems. In CLP, when reducing a goal, unification can also lead to functional constraints. For example, when matching $p(Z^2 + 1)$ with a rule on $p(X)$ where both X and Z are finite domain variables, a functional constraint $X = Z^2 + 1$ is produced.

Most work on functional constraints follows the approach in CSP which is based on arc or path consistency [11, 3]. In this paper, we propose a new method — *variable substitution* — to process functional constraints. The idea is that if a constraint is functional on a variable, this variable in another constraint can be substituted using the functional constraint without losing any solution.

Given a variable, the *variable elimination* method substitutes this variable in *all* constraints involving it such that it is effectively "eliminated" from the problem. This idea is applied to reduce any problem containing non-functional

* Part of this work was supported by National Univ. of Singapore, grant 252-000-303-112.

constraints into a canonical form where some variables can be safely ignored when solving the problem. We design an efficient algorithm to reduce, in $\mathcal{O}(ed^2)$ where e is the number of constraints and d the size of the largest domain of the variables, a general binary CSP containing functional constraints into a canonical form. This reduction simplifies the problem and makes the functional portion trivially solvable. When the functional constraints are also bi-functional, then the algorithm is linear in the size of the CSP.

Many CLP systems with finite domains make use of constraint propagation algorithms such as arc consistency. Our experiments show that the substitution based "global" treatment of functional constraints can significantly speed up propagation based solvers.

In the rest of the paper, background on CSPs and functional constraints is given in Section 2. Variable substitution for binary functional constraints is introduced and studied in Section 3. Section 4 presents results on algorithms for variable elimination in general CSPs containing functional constraints. Section 5 presents an experimental study. Functional elimination is extended to non-binary constraints in Section 6. Related work is discussed in Section 7 and concluded in Section 8.

2 Preliminaries

We begin with the basic concepts and notation used in this paper.

A binary *Constraint Satisfaction Problem (CSP)* (N, D, C) consists of a finite set of variables $N = \{1, \cdots, n\}$, a set of domains $D = \{D_1, \cdots, D_n\}$, where D_i is the domain of variable i , and a set of constraints each of which is a binary relation between two variables in N.

A constraint between two variables i and j is denoted by c_{ij}. Symbols a and b possibly with subscript denote the values in a domain. A constraint c_{ij} is a set of allowed tuples. We assume testing whether a tuple belongs to a constraint takes constant time. For $a \in D_i$ and $b \in D_j$, we use either $(a, b) \in c_{ij}$ or $c_{ij}(a, b)$ to denote that a and b satisfy the constraint c_{ij}. For the problems of interest here, we require that for all $a \in D_i$ and $b \in D_j$, $(a, b) \in c_{ij}$ if and only if $(b, a) \in c_{ji}$. If there is no constraint on i and j, c_{ij} denotes a universal relation, i.e., $D_i \times D_j$.

A constraint graph $G = (V, E)$, where $V = N$ and $E = \{\{i, j\} \mid \exists c_{ij} \in C\}$, is usually used to describe the topological structure of a CSP. A *solution* of a constraint satisfaction problem is an assignment of a value to each variable such that the assignment satisfies all the constraints in the problem. A CSP is *satisfiable* if it has a solution. The *solution space* of a CSP is the set of all its solutions. Two CSPs are *equivalent* if and only if they have the same solution space. Throughout this paper, n represents the number of variables, d the size of the largest domain of the variables, and e the number of constraints in C.

We need two operations on constraints in this paper. One is the intersection of two constraints (intersection of the sets of tuples) that constrain the same set of variables. The other operation is the composition, denoted by the symbol "$\circ$,"

of two constraints sharing a variable. The composition of two relations is:

$$c_{jk} \circ c_{ij} = \{(a, c) \mid \exists b \in D_j, \textit{such that } (a, b) \in c_{ij} \wedge (b, c) \in c_{jk}\}.$$

Composition is a basic operation in our variable substitution method. Composing c_{ij} and c_{jk} gives a new constraint on i and k.

Example Consider constraints $c_{ij} = \{(a_1, b_1), (a_2, b_2), (a_2, b_3)\}$ and $c_{jk} = \{(b_1, c_1), (b_2, c_2), (b_3, c_2)\}$. The composition of c_{ij} and c_{jk} is a constraint on i and k: $c_{ik} = \{(a_1, c_1), (a_2, c_2)\}$. □

A constraint c_{ij} is *functional* on j if for any $a \in D_i$ there exists at most one $b \in D_j$ such that $c_{ij}(a, b)$. c_{ij} is *functional* on i if c_{ji} is functional on i. Given a constraint c_{ij} functional on j and a value $a \in D_i$, we assume throughout the paper that in constant time we can find the value $b \in D_j$, if there is one, such that $(a, b) \in c_{ij}$.

A special case of functional constraints are equations. These are ubiquitous in CLP. A typical functional constraint in arithmetic is a binary linear equation like $2x = 5 - 3y$ which is functional on x and on y. Functional constraints do not need to be linear. For example, a nonlinear equation $x^2 = y^2$ where $x, y \in 1..10$ is also functional on both x and y. In scene labeling problems [7], there are many functional constraints and other special constraints. □

When a constraint c_{ij} is functional on j, for simplicity, we say c_{ij} is functional by making use of the fact that the subscripts of c_{ij} are an ordered pair. When c_{ij} is functional on i, c_{ji} is said to be functional. That c_{ij} is functional does not mean c_{ji} is functional. In this paper, the definition of functional constraints is different from the one in [12, 11] where constraints are functional on each of its variables, leading to the following notion.

A constraint c_{ij} is *bi-functional* if c_{ij} is functional on i and on j. A bi-functional constraint is called *bijective* in [3]. For functional constraints, we have the following property on their composition and intersection: 1) If c_{ij} and c_{jk} are functional on j and k respectively, their composition remains functional; and 2) The intersection of two functional constraints remains functional.

3 Variable Substitution and Elimination Using Binary Functional Constraints

We introduce the idea of variable substitution. Given a CSP (N, D, C), a constraint $c_{ij} \in C$ that is functional on j, and a constraint c_{jk} in C, we can substitute i for j in c_{jk} by composing c_{ij} and c_{jk}. If there is already a constraint $c_{ik} \in C$, the new constraint on i and k is simply the intersection of c_{ik} and $c_{jk} \circ c_{ij}$.

Definition 1. *Consider a CSP (N, D, C), a constraint $c_{ij} \in C$ functional on j, and a constraint $c_{jk} \in C$. To* substitute i for j *in c_{jk}, using c_{ij}, is to get a new CSP where c_{jk} is replaced by $c'_{ik} = c_{ik} \cap (c_{jk} \circ c_{ij})$. The variable i is called the* substitution variable.

A fundamental property of variable substitution is that it preserves the solution space of the problem.

Property 1. Given a CSP (N, D, C), a constraint $c_{ij} \in C$ functional on j, and a constraint $c_{jk} \in C$, the new problem obtained by substituting i for j in c_{jk} is equivalent to (N, D, C).

Proof Let the new problem after substituting i for j in c_{jk} be (N, D, C') where $C' = (C - \{c_{jk}\}) \cup \{c'_{ik}\}$ and $c'_{ik} = c_{ik} \cap (c_{jk} \circ c_{ij})$.

Assume $(a_1, a_2, \cdots, a_n)$ is a solution of (N, D, C). We need to show that it satisfies C'. The major difference between C' and C is that C' has new constraint c'_{ik}. It is known that $(a_i, a_j) \in c_{ij}$, $(a_j, a_k) \in c_{jk}$, and if there is c_{ik} in C, $(a_i, a_k) \in c_{ik}$. The fact that $c'_{ik} = (c_{jk} \circ c_{ij}) \cap c_{ik}$ implies $(a_i, a_k) \in c'_{ik}$. Hence, c'_{ik} is satisfied by $(a_1, a_2, \cdots, a_n)$.

Conversely, we need to show that any solution $(a_1, a_2, \cdots, a_n)$ of (N, D, C') is a solution of (N, D, C). Given the difference between C' and C, it is sufficient to show the solution satisfies c_{jk}. We have $(a_i, a_j) \in c_{ij}$ and $(a_i, a_k) \in c'_{ik}$. Since $c'_{ik} = (c_{jk} \circ c_{ij}) \cap c_{ik}$, there must exist $b \in D_j$ such that $(a_i, b) \in c_{ij}$ and $(b, a_k) \in c_{jk}$. As c_{ij} is functional, b has to be a_j. Hence, a_j and a_k satisfy c_{jk}. $\square$

Based on variable substitution, we can eliminate a variable from a problem so that no constraint will be on this variable (except the functional constraint used to substitute it).

Definition 2. *Given a CSP (N, D, C) and a constraint $c_{ij} \in C$ functional on j, to* eliminate j using c_{ij} *is to substitute i for j, using c_{ij}, in every constraint $c_{jk} \in C$ (except c_{ji}).*

We can also substitute i for j in c_{ji} to obtain c'_{ii} and then intersect c'_{ii} with the identity relation on D_i, equivalent to a direct revision of the domain of i with respect to c_{ij}. This would make the algorithms presented in this paper more uniform, i.e., only operations on constraints are used. Since in most algorithms we want to make domain revision explicit, we choose not to substitute i for j in c_{ji}.

Given a functional constraint c_{ij} of a CSP (N, D, C), let C_j be the set of all constraints involving j, except c_{ij}. The elimination of j using c_{ij} results in a new problem (N, D, C') where

$$C' = (C - C_j) \cup \{c'_{ik} \mid c'_{ik} = (c_{jk} \circ c_{ij}) \cap c_{ik}, c_{jk} \in C\}.$$

In the new problem, there is only one constraint c_{ij} on j and thus j can be regarded as being "eliminated".

Example Consider a problem with three constraints whose constraint graph is shown in Figure 1 (a). Let c_{ij} be functional. The CSP after j has been eliminated using c_{ij} is shown in Figure 1 (b). In the new CSP, constraints c_{jk} and c_{jl} are discarded, and new constraints $c_{ik} = c_{jk} \circ c_{ij}$ and $c_{il} = c_{jl} \circ c_{ij}$ are added. $\square$

The variable elimination involves "several" substitutions and thus preserves the solution space of the original problem by Property 1.

Property 2. Given a CSP (N, D, C) and a functional constraint $c_{ij} \in C$, the new problem (N, D, C') obtained by the elimination of variable j using c_{ij} is equivalent to (N, D, C).

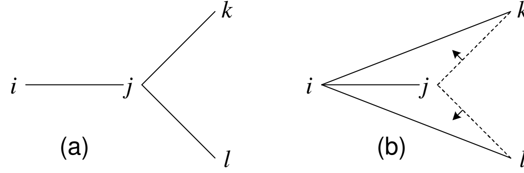

Fig. 1. (a): A CSP with a functional constraint c_{ij}. (b): The new CSP after eliminating the variable j using c_{ij}.

4 Elimination Algorithms for CSPs with Functional Constraints and Non-Functional Constraints

We now extend variable elimination to general CSPs with functional and non-functional constraints. The idea of variable elimination (Definition 2 in Section 3) can be used to reduce a CSP to the following canonical functional form.

Definition 3. *A CSP (N, D, C) is in* canonical functional form *if for any constraint $c_{ij} \in C$ functional on j, the following conditions are satisfied: 1) if c_{ji} is also functional on i(i.e., c_{ij} is bi-functional), either i or j is not constrained by any other constraint in C; 2) otherwise, j is not constrained by any other constraint in C.*

As a trivial example, a CSP without any functional constraint is in canonical functional form. If a CSP contains some functional constraints, it is in canonical functional form intuitively if for any functional constraint c_{ij}, there is only one constraint on j. As an exception, the first condition in the definition implies that when c_{ij} is bi-functional, one variable of $\{i, j\}$ might have several bi-functional constraints on it.

In a canonical functional form CSP, *the functional constraints form disjoint star graphs*. A *star graph* is a tree where there exists a node, called the *center*, such that there is an edge between this center node and every other node. We call the variable at the center of a star graph, a *free variable*, and other variables in the star graph *eliminated variables*. Fig. 1(b) is a star graph, assuming c_{jk} and c_{jl} are functional on k and l respectively, with free variable i. The constraint between a free variable i and an eliminated variable j is functional on j, but it may or may not be functional on i. In the special case that the star graph contains only two variables i and j and c_{ij} is bi-functional, any one of the variables can be called a free variable while the other is called an eliminated variable.

If a CSP is in canonical functional form, all functional constraints and the eliminated variables can be *ignored* when we try to find a solution for this problem. Thus, to solve a CSP (N, D, C) in canonical functional form whose non-eliminated variables are NE, we only need to solve a smaller problem (NE, D', C') where D' is the set of domains of the variables NE and $C' = \{c_{ij} \mid c_{ij} \in C \text{ and } i, j \in NE\}$.

Proposition 1. *Consider a CSP $P_1 = (N, D, C)$ in a canonical functional form and a new CSP $P_2 = (NE, D', C')$ formed by ignoring the eliminated variables in P_1. For any free variable $i \in N$ and any constraint $c_{ij} \in C$ functional on j, assume any value of D_i has a support in D_j and this support can be found in constant time. Any solution of P_2 is extensible to a unique solution of P_1 in $\mathcal{O}(|N - NE|)$ time. Any solution of P_1 can be obtained from a solution of P_2.*

Proof Let $(a_1, a_2, \cdots, a_{|NE|})$ be a solution of (NE, D', C'). Consider any eliminated variable $j \in N - NE$. In C, there is only one constraint on j. Let it be c_{ij} where i must be a free variable. By the assumption of the proposition, the value of i in the solution has a unique support in j. This support will be assigned to j. In this way, a unique solution for (N, D, C) is obtained. The complexity of this extension is $\mathcal{O}(|N - NE|)$.

Let S be a solution of (N, D, C) and S' the portion of S restricted to the variables in NE. S' is a solution of (NE, D', C') because $C' \subseteq C$. The unique extension of S' to a solution of P_1 is exactly S. □

Any CSP with functional constraints can be transformed into canonical functional form by variable elimination using the algorithm in Fig. 2. Given a constraint c_{ij} functional on j, Line 1 of the algorithm substitutes i for j in all constraints involving j. Note the arc consistency on c_{ik}, for all neighbor k of i, is enforced by line 3.

---

algorithm Variable-Elimination(**inout** (N, D, C), **out** *consistent*) {
    $L \leftarrow N$;
    while (There is $c_{ij} \in C$ functional on j where $i, j \in L$ and $i \neq j$){
        // Eliminate variable j,
1. $C \leftarrow \{c'_{ik} \mid c'_{ik} \leftarrow (c_{jk} \circ c_{ij}) \cap c_{ik}, c_{jk} \in C, k \neq i\} \cup (C - \{c_{jk} \in C \mid k \neq i\})$;
2. $L \leftarrow L - \{j\}$;
3.   Revise the domain of i wrt c_{ik} for every neighbour k of i;
        if (D_i is empty) **then** { *consistent* $\leftarrow$ **false**; return }
    }
    consistent $\leftarrow$ **true**;
}

Fig. 2. A variable elimination algorithm to transform a CSP into a canonical functional form.

Theorem 1. *Given a CSP (N, D, C),* `Variable-Elimination` *transforms the problem into a canonical functional form in $\mathcal{O}(n^2 d^2)$.*

Proof Assume `Variable-Elimination` transforms a CSP $P_1 = (N, D, C)$ into a new problem $P_2 = (N, D', C')$. We show that P_2 is of canonical functional form. For any variable $j \in N$, if there is a constraint $c_{ij} \in C'$ functional on j, there are two cases. Case 1: $j \notin L$ when the algorithm terminates. This

means that c_{ij} is the functional constraint that is used to substitute j in other constraints (Line 1). After substitution, c_{ij} is the unique constraint on j. Case 2: $j \in L$ when the algorithm terminates. Variable i must not be in L (otherwise, j will be substituted by Line 1). This implies that i was substituted using c_{ji}. Thus, c_{ji} is the only functional constraint on i in P_2. Hence, c_{ij} is bi-functional and i is not constrained by any other constraints.

Next, we show the complexity of `Variable-Elimination`. It eliminates any variable in N at most once (Line 2). For each variable j and a constraint c_{ij} functional on j, there are at most $n-2$ other constraints on j. The variable j in those constraints needs to be substituted. The complexity of the substitution for each constraint is $\mathcal{O}(d^2)$. The elimination of j (Line 1) takes $\mathcal{O}(nd^2)$. There are at most $n-1$ variables to eliminate and thus the worst case complexity of the algorithm is $\mathcal{O}(n^2d^2)$. □

It is worth noting that the variable elimination algorithm is able to globally solve some CSPs containing non-functional constraints.

Example Consider a simple example where there are three variables i, j, and k whose domains are $\{1, 2, 3\}$ and the constraints are $i = j$, $i = k+1$, and $j \neq k$. Note that although the constraints are listed in an equational form, the actual constraints are explicit and discrete, thus normal equational reasoning might not be applicable. By eliminating j using c_{ij}, c_{ik} becomes $\{(2, 1), (3, 2)\}$, and the domain of i becomes $\{2, 3\}$. The non-functional constraint c_{jk} is gone. The problem is in canonical functional form. A solution can be obtained by letting i be 2 and consequently $j = 2$ and $k = 1$. □

By carefully choosing an ordering of the variables to eliminate, a faster algorithm can be obtained. The intuition is that once a variable i is used to substitute for other variables, i itself should not be substituted by any other variable later.

Example Consider a CSP with functional constraints c_{ij} and c_{jk}. Its constraint graph is shown in Fig. 3(a) where a functional constraint is represented by an arrow. If we eliminate k and then j, we first get c_{jl_1} and c_{jl_2}, and then get c_{il_1} and c_{il_2}. Note that j is first used to substitute for k and later is substituted by i. If we eliminate j and then k, we first get c_{ik}, and then get c_{il_1} and c_{il_2}. In this way, we reduce the number of compositions of constraints. □

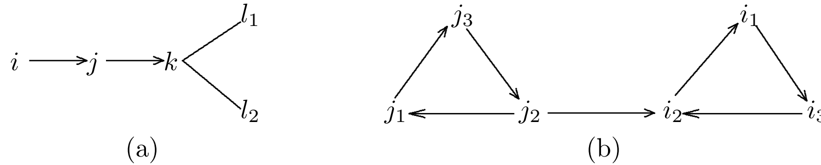

Fig. 3. (a) The constraint graph of a CSP with functional constraints c_{ij} and c_{jk}. (b) A directed graph.

Given a CSP $P = (N, D, C)$, P^F is used to denote its directed graph (V, E) where $V = N$ and $E = \{(i, j) \mid c_{ij} \in C \text{ and } c_{ij} \text{ is functional on } j\}$. Non-functional constraints in C do not appear in P^F. A subgraph of a directed graph

is *strongly connected* if for any two vertices of the subgraph, any one of them is reachable from the other. A *strongly connected component* of a directed graph is a maximum subgraph that is strongly connected. To describe our algorithm we need the following notation.

Definition 4. *Given a directed graph (V, E), a sequence of the nodes of V is a* functional elimination ordering *if for any two nodes i and j, i before j in the sequence implies that there is a path from i and j. A* functional elimination ordering *of a CSP problem P is a functional elimination ordering of P^F.*

The functional elimination ordering is used to overcome the redundant computation shown in the example on Fig. 3(a). Given a directed graph G, a functional elimination ordering can be found by: 1) finding all the strongly connected components of G; 2) modifying G by taking every component as one vertex with edges changed and/or added accordingly; 3) finding a topological ordering of the nodes in the new graph; and 4) replacing any vertex v in the ordering by any sequence of the vertices of the strongly connected component represented by v.

To illustrate the process, consider the example in Fig. 3(b) which can be taken as P^F for some CSP problem P. All strongly connected components are $\{j_1, j_2, j_3\}$, denoted by c_1, and $\{i_1, i_2, i_3\}$, denoted by c_2. We construct the new graph by replacing the components by vertices: $(\{c_1, c_2\}, \{(c_1, c_2)\})$. We have the edge (c_1, c_2) because the two components are connected by (j_2, i_2). The topological ordering of the new graph is $\langle c_1, c_2 \rangle$. Now we can replace c_1 by any sequence of j's and c_2 by any sequence of i's. For example, we can have a functional elimination ordering $\langle j_3, j_2, j_1, i_2, i_3, i_1 \rangle$.

The algorithm `Linear-Elimination` in Fig. 4 first finds a functional elimination ordering (Line 1). Line 4 and 6 are to *process* all the variables in O. Every variable i of O is *processed* as follows: i will be used to substitute for all the variables *reachable* from i *through constraints that are functional in C^0 and still exist in the current C*. Those constraints are called *qualified* constraints. Specifically, L initially holds the immediate reachable variables through qualified constraints (Line 8). Line 9 is a loop to eliminate all variables reachable from i. The loop at Line 11 is to eliminate j using i from the current C. In this loop, if a constraint c_{jk} is qualified (Line 14), k is reachable from i through qualified constraints. Therefore, it is put into L (Line 15).

To illustrate the ideas underlying the algorithm, consider the example in Fig. 3(b). Now, we assume the edges in the graph are the only constraints in the problem. Assume the algorithm finds the ordering given earlier: $O = \langle j_3, j_2, j_1, i_2, i_3, i_1 \rangle$. Next, it starts from j_3. The qualified constraints leaving j_3 are $c_{j_3 j_2}$ only. So, the immediate reachable variables through qualified constraints are $L = \{j_2\}$. Take and delete j_2 from L. Substitute j_3 for j_2 in constraints $c_{j_2 i_2}$ and $c_{j_2 j_1}$. As a result, constraints $c_{j_2 i_2}$ and $c_{j_2 j_1}$ are removed from C while $c_{j_3 j_1} = c_{j_3 j_1} \cap (c_{j_2 j_1} \circ c_{j_3 j_2})$ and new constraint $c_{j_3 i_2} = c_{j_2 i_2} \circ c_{j_3 j_2}$ is introduced to C. One can verify that both $c_{j_2 j_1}$ and $c_{j_2 i_2}$ are qualified. Hence, variables j_1 and i_2 are reachable from j_3 and thus are put into L. Assume j_1 is selected from L. Since there are no other constraints on j_1, nothing is done. Variable i_2

```
algorithm Linear-Elimination(inout (N, D, C)) {
 1. Find a functional elimination ordering O of the problem;
 2. Let C^0 be C; any c_ij in C^0 is denoted by c^0_ij;
 3. For each i ∈ N, it is marked as not eliminated;
 4. while (O is not empty) {
        Take and delete the first variable i from O;
 6.     if (i is not eliminated) {
 8.         L ← {j | (i, j) ∈ C and c^0_ij is functional};
 9.         while (L not empty) {
                Take and delete j from L;
11.             for any c_jk ∈ C − {c_ji} { // Substitute i for j in c_jk;
                    c'_ik ← c_jk ∘ c_ij ∩ c_ik;
                    C ← C ∪ {c'_ik} − {c_jk};
14.                 if (c^0_jk is functional) then
15.                     L ← L ∪ {k};
                }
16.             Mark j as eliminated;
            } // loop on L
        }
    } // loop on O
} // end of algorithm
```

Fig. 4. A variable elimination algorithm of complexity $O(ed^2)$.

is then selected from L. By eliminating i_2 using j_3, $c_{i_2 i_1}$ and $c_{i_2 i_3}$ are removed from C and $c_{j_3 i_1}$ and $c_{j_3 i_3}$ are added to C. Constraint $c_{i_2 i_1}$ is qualified, and thus i_1 is added to L. Note that $c_{i_2 i_3}$ is not qualified because it is not functional on i_3 in terms of the graph. We take out the only variable i_1 in L. After i_1 is eliminated using j_3, $c_{i_1 i_3}$ is removed from C, and constraint $c_{j_3 i_3}$ is updated to be $c_{j_3 i_3} \cap (c_{i_1 i_3} \circ c_{j_3 i_1})$. Since $c_{i_1 i_3}$ is qualified, i_3 is added to L. One can see that although i_3 was not reachable when i_2 was eliminated, it finally becomes reachable because of i_1. In general, all variables in a strongly connected component are reachable from the variable under processing if one of them is reachable. Now, take i_3 out of L, and nothing is done because there are no other constraints incident on it. Every variable except j_3 is marked as *eliminated* (Line 16), the **while** loop on O (Line 4 and 6) terminates.

Theorem 2. *Given a CSP problem, the worst case time complexity of* `Linear-Elimination` *is* $O(ed^2)$ *where* e *is the number of constraints and* d *the size of maximum domain in the problem.*

Proof To find a functional elimination ordering involves the identification of strongly connected components and topological sorting. Each of two operations takes linear time. Therefore, Line 1 of the algorithm takes $\mathcal{O}(n + e)$.

The **while** loop of Line 4 takes $\mathcal{O}(ed^2)$. Assume that there is a unique identification number associated with each constraint in C. After some variable of a

constraint is substituted, the constraint's identification number refers to the new constraint. For any identification number α, let its first associated constraint be c_{jk}. Assuming j is substituted by some other variable i, we can show that i will be never be substituted later in the algorithm. By the algorithm, i is selected at Line 6. So, all variables before i in O have been processed. Since i is not eliminated, it is not reachable from any variable before it (in terms of O) through qualified constraints (due to loop of Line 9). Hence, there are two cases: 1) there is no constraint c_{mi} of C such that c^0_{mi} is functional on i, 2) there is at least one constraint c_{mi} of C such that c^0_{mi} is functional on i. In the first case, our algorithm will never substitute i by any other variable. By definition of functional elimination ordering, case 2 implies that i belongs to a strongly connected component whose variables have not been eliminated yet. Since all variables in the component will be substituted by i, after the loop of Line 9, there is no constraint c_{mi} of C such that c^0_{mi} is functional on i. Hence, i will never be substituted. In a similar fashion, if variable k is substituted by l, l will never be substituted later by the algorithm. So, there are at most two substitutions occurring to α. By definition, substitution involves a functional constraint, its complexity is $O(d^2)$ in the worst case. Since there is a unique identification number for each constraint, the time taken by **while** loop at Line 4 is $O(ed^2)$.

In summary, the worst case time complexity of the algorithm is $O(ed^2)$. □

To characterize the property of `Linear-Elimination`, we need the following notation.

Definition 5. *Given a problem P, let C^0 be the constraints before* `Linear-Elimination` *and C the constraints of the problem at any moment during the algorithm. A constraint c_{ij} of C is* trivially functional *if it is functional and satisfies the condition: c^0_{ij} is functional or there is a path $i_1(=i), i_2, \cdots, i_m(=j)$ in C_0 such that $c^0_{i_k i_{k+1}} (k \in 1..m-1)$ is functional on i_{k+1}.*

Theorem 3. *Algorithm* `Linear-Elimination` *transforms a CSP (N, D, C) into a canonical functional form if all newly produced functional constraints (due to substitution) are trivially functional.*

The proof of this result is straightforward and thus omitted here.

Corollary 1. *For a CSP problem with non-functional constraints and bi-functional constraints, the worst case time complexity of algorithm* `Linear-Elimination` *is linear to the problem size.*

This result follows the observation below. When the functional constraint involved in a substitution is bi-functional, the complexity of the composition is linear to the constraints involved. From the proof of Theorem 2, the complexity of the algorithm is linear to the size of all constraints, i.e., the problem size.

Corollary 2. *Consider a CSP with both functional and non-functional constraints. If there is a variable of the problem such that every variable of the CSP is reachable from it in P^F, the satisfiability of the problem can be decided in $\mathcal{O}(ed^2)$ using* `Linear-Elimination`.

For a problem with the property given in the corollary, its canonical functional form becomes a star graph. So, any value in the domain of the free variable is extensible to a solution if we add (arc) consistency enforcing during `Linear-Elimination`. The problem is not satisfiable if a domain becomes empty during the elimination process.

5 Experimental Results

We investigate to see the effectiveness of variable elimination on problem solving. In our experiments, a problem is either directly solved by a general solver or variable elimination is invoked before the solver.

Since there are no publicly available benchmarks on functional constraints, we generate random problems $\langle n, d, e, nf, t \rangle$ where n is the number of variables, d domain size, e the number of constraints, *nf* the number of functional constraints, and t the tightness of non-functional constraints. Tightness r is defined as the percentage of allowed tuples over d^2. Except the *nf* functional constraints, all other constraints are non-functional. Each functional constraint is made to have d allowed tuples. Our implementation allows other tightness for functional constraints. However, we observed from the experiments that if we make the number of allowed tuples less than d, the problems are easy (i.e., with very few backtracks) to solve.

We selected meaningful problems from the ones generated to do benchmarking. In the context of random problems, the tightness $1/d$ of functional constraints is rather tight. Therefore, when we increase *nf*, the "hardness" of the problems drops correspondingly. In our experiments, we systematically examine the problems with the following setting: n, d are 50, e varies from 100 to 710 with step size 122 (10% of all possible constraints), *nf* varies from 2 to 12, and t varies from 0.2 to 1.0 with step size 0.05. When *nf* is small (e.g, 2), there are so many hard problems that we can only scan a small portion of the problems (we stop running when the time limit $4 * 10^4$s is reached). When *nf* is large (e.g., 12), even for the most difficult problem instances, the number of backtracks is small and thus they are simple. For example, when $nf = 12$, the most difficult problems are with $e = 710$. The table below shows the hardness of the problems, with $nf = 12$ and $e = 710$, in terms of number of backtracks (#bt) needed. When t is from 0.2 to 0.65, #bt is 0. For the most difficult case of t being 0.8, #bt is small (around 1000). On the other hand, when *nf* is small, one can expect that the application of elimination may not make much difference.

t	$0.2 - 0.65$	0.7	0.75	0.8	0.85	$0.9 - 0.95$
#bt	0	5.7	22.9	1023	0.2	0

Due to the observations above, we evaluate the algorithm on non-trivial cases (e.g., trivial cases include few backtracks or very small number of functional constraints). We study the effectiveness of variable elimination for each *nf* on the most difficult problems as we discovered in the exploration process above. The results with *nf* varying from 6 to 12 are shown in Fig. 5. The results were

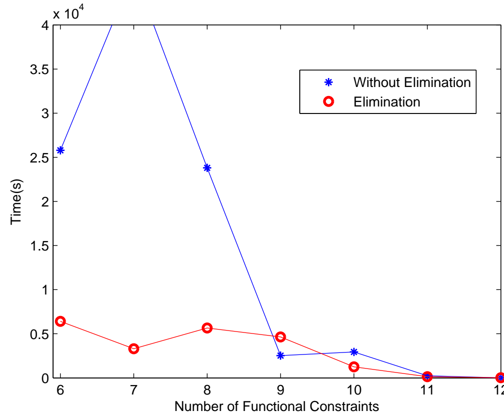

Fig. 5. Performance of the algorithms on random problems. When $nf = 7$, time limit is reached by the solver without variable elimination.

obtained on a DELL PowerEdge 1850 (two 3.6GHz Intel Xeon CPUs) in Linux. We implement both the elimination algorithm and a general solver in C++. The solver uses the standard backtracking algorithm. During the search, a variable with maximum degree is selected first with tie broken by lexicographical order while the value of a variable is selected in a lexicographical order.

For the problem instances used in Figure 5, the time to transform the instances into their canonical forms is negligible compared to the time solving the problems. There are two reasons. First, the number of constraints involved in the elimination is relatively small compared to the total number of constraints in the problems. Second, the algorithm is as efficient as the optimal general arc consistency algorithm used in the solver. Thirdly, the elimination is applied only once before the backtracking search.

The results show that the variable elimination can significantly speed up problem solving by more than 5 times on difficult problems where a lot of backtracks occur. As the number of functional constraints increases, one would assume that the variable elimination should be more effective. However, we notice that as the number of functional constraints increases, the random problems become simpler, which may decrease the benefit of elimination. For example, when $nf = 12$ (see the table above), to solve the problems, the solver only needs about a thousand backtracks. In this case, the variable elimination will not be able to save much. We notice that when $nf = 9$, the variable elimination approach is slower. There are two possible explanations. The first is that variable elimination changes the topology of the problem, which may affect the effectiveness of the heuristics of the general solver. The second is that we use only 10 problem instances per configuration which may cause some unstable results. We have also manually tried many configurations. The results show very similar trend to the one in Fig. 5.

Remark. Views which are used in several existing CP systems (e.g., [9]) can be thought of as an efficient way to enforce arc consistency on bi-functional constraints. We used our own solver rather than one with views due to the following reason. The percentage of functional constraints in our problem instances is less than 2%. Improving the arc consistency efficiency on them won't affect the overall performance too much.

6 Variable Elimination and Non-binary Constraints

Non-binary constraints such as arithmetic or global constraints are common in CP systems. We discuss how variable elimination of functional constraints can be applied to these constraints.

Non-binary constraints are either extensional (defined explicitly) or intensional (defined implicitly). To substitute a variable in an extensional non-binary constraints, we can define the composition of a non-binary constraint with a binary constraint as a straightforward generalization of the composition operation defined in Section 2.

For intentional constraints, there are usually particular propagators with specific algorithm for the constraint. We sketch below an approach which allows variable elimination to be employed with generic propagators. Assume we have a linear constraint c_1: $ax + by + cz < d$ and a constraint c_{wy} functional on y. To substitute y in c_1, we simply modify c_1 to be $ax + bw + cz < d$ and mark w as a *shadow variable* (w needs special treatment by the propagator, which will be clear later). We call y the *shadowed* variable. Assume we also have c_{uw} functional on w. To eliminate w, c_1 is further changed to $ax + bu + cz < d$. Since w is a shadow variable, we generate a new constraint c_{uy} using c_{uw} and c_{wy} in a standard way as discussed in this paper. Now u becomes the shadow variable while the shadowed variable is still y (variable w is gone). Assume we need to make c_1 arc consistency. First "synchronize the domains" of y and u using c_{uy}, i.e., enforce arc consistency on c_{uy}. (Note that due to elimination, c_{wy} and c_{uw} are no longer involved in constraint solving). Next, we enforce arc consistency on c_1. During the process, since u is a shadow variable, all domain operations are on y instead of u. After making c_1 arc consistent, synchronize the domain of y and u again. (If the domain of u is changed, initiate constraint propagation on constraints involving u.) This approach is rather generic: for any intensional constraints, synchronize the domains of the shadow variables and shadowed variables, apply whatever propagation methods on the shadowed variables (and other non-shadow variables), synchronize the domains of shadow variables and shadowed variables again. In fact, the synchronization of the domains of the shadow and shadowed variables (e.g., u and y above) can be easily implemented using the concept of views [9].

7 Related Work

Bi-functional constraints have been studied in the context of arc consistency (AC) algorithms since Van Hentenryck et al. [11] proposed a worst case optimal AC algorithm with $O(ed)$, which is better than the time complexity ($\mathcal{O}(ed^2)$) of optimal AC algorithms such as AC2001/3.1 [2] for arbitrary binary constraints. Liu [8] proposed a fast AC algorithm for a special class of *increasing* bi-functional constraints. Affane and Bennaceur [1] introduced a new type of consistency, label-arc consistency, and showed that the bi-functional constraints with limited extensions to other constraints can be (globally) solved, but no detailed analysis of their algorithms is given. In [12], we proposed a variable elimination method to solve *bi-functional* constraints in $\mathcal{O}(ed)$. Functional constraints are not discussed in those works.

David introduced *pivot consistency* for binary functional constraints in [3]. Both pivot consistency and variable substitution help to reduce a CSP into a special form. There are some important differences between pivot consistency and variable substitution. First, the concept of pivot consistency, a special type of directional path consistency, is quite complex. It is defined in terms of a variable ordering, path (of length 2) consistency, and concepts in directed graphs. Variable substitution is a much simpler concept as shown in the paper. For both binary and non-binary CSPs, the concept of variable substitution is intuitive and simple. Next, by the definition of pivot consistency, to make a CSP pivot consistent, there must be a certain functional constraint on each of the *non-root* variables. Variable substitution is more flexible. It can be applied whenever there is a functional constraint in a problem. Finally, to reduce a problem, the variable elimination algorithm takes $\mathcal{O}(ed^2)$ while pivot consistency algorithm takes $\mathcal{O}((n^2 - r^2)d^2)$, where r is the number of *root* variables.

Another related approach is bucket elimination [4]. The idea in common behind bucket elimination and variable substitution is to exclude the impact of a variable on the whole problem. The difference between them lies in the way variable elimination is performed. In each elimination step, substitution does not increase the arity of the constraints while bucket elimination could generate constraints with *higher arity* (possibly exponential space complexity). The former may generate more constraints than the latter, but it will *not* increase the total number of constraints in the problem.

CLP [6] systems often make use of variable substitution and elimination. The classic unification algorithm is a good example. A more complex example is CLP($\mathcal{R}$) [5] which has constraints on finite trees and arithmetic. Variables in arithmetic constraints are substituted out using a parametric normal form which is applied during unification and also when solving arithmetic constraints. Our approach is compatible with such CLP solvers which reduce the constraint store to a normal form using variable substitution. We remark that any CLP language or system which has finite domain constraints or CSP constraints will deal with bi-functional constraints because of unification. Thus, a variable substitution approach will actually be more powerful than just simple finite domain propagation on equations.

8 Conclusion

We have introduced a variable substitution method to reduce a problem with both functional and non-functional constraints. Compared with the previous work on bi-functional and functional constraints, the new method is not only conceptually simple and intuitive but also reflects the fundamental property of functional constraints. Our experiments also show that variable elimination can significantly improve the performance of a general solver in dealing with functional constraints.

For a binary CSP with both functional and non-functional constraints, an algorithm is presented to transform it into a canonical functional form in $\mathcal{O}(ed^2)$. This leads to a substantial simplification of the CSP with respect to the functional constraints. In some cases, as one of our results (Corollary 2) shows, the CSP is already solved. Otherwise, the canonical form can be solved by ignoring the eliminated variables. For example, this means that search only needs to solve a smaller problem than the one before variable substitution (or elimination).

References

1. Affane, M.S., Bennaceur, H.: A Labelling Arc Consistency Method for Functional Constraints. In: Freuder, E.C. (ed) CP 96. LNCS, vol. 1118, pp. 16–30. Springer, Heilderberg (1996)
2. Bessiere, C., Regin, J.C., Yap, R.H.C., Zhang, Y.: An Optimal Coarse-grained Arc Consistency Algorithm. Artificial Intelligence 165(2), 165–185 (2005)
3. David, P.: Using Pivot Consistency to Decompose and Solve Functional CSPs. J. of Artificial Intelligence Research 2, 447–474 (1995)
4. Dechter, R.: Bucket elimination: A Unifying Framework for Reasoning. Artificial Intelligence 113, 41–85 (1999)
5. Jaffar, J., Michaylov, S., Stuckey, P.J., Yap, R.H.C.: The CLP($\mathcal{R}$) Language and System. ACM Trans. on Programming Languages and Systems 14(3), 339–395 (1992)
6. Jaffar, J., Maher, M.J.: Constraint Logic Programming. J. of Logic Programming 19/20, 503–581 (1994)
7. Kirousis, L.M.: Fast Parallel Constraint Satisfaction. Artificial Intelligence 64, 147–160 (1993)
8. Liu, B.: Increasing Functional Constraints Need to be Checked Only Once. In: IJCAI-95, pp. 119–125. Morgan Kaufmann, San Francisco (1995)
9. Schulte, C., Tack, G.: Views and Iterators for Generic Constraint Implementations. In: Recent Advances in Constraints. LNCS, vol. 4651, pp. 37–48. Springer, Heilderberg (2005)
10. Stallman, R.M., Sussman, G.J.: Forward Reasoning and Dependency-directed Backtracking in a System for Computer-aided Circuit Analysis. Artificial Intelligence 9(2), 135–196 (1977)
11. Van Hentenryck, P., Deville, Y., Teng, C.M.: A Generic Arc-consistency Algorithm and its Specializations. Artificial Intelligence 58, 291–321, 1992.
12. Zhang, Y., Yap, R.H.C., Jaffar, J.: Functional Elimination and 0/1/All Constraints. In: AAAI-99, pp. 275–281. AAAI Press, Menlo Park (1999)