Constraint Programming and Graph Algorithms

Michel Gangnet
Paris Research Laboratory
Digital Equipment Corporation
Rueil-Malmaison, France

Burton Rosenberg
Department of Mathematics and
Computer Science
University of Miami

Abstract

A *constraint system* includes a set of variables and a set of relations among these variables, called *constraints*. The solution of a constraint system is an assignment of values to variables so that all, or many, of the relations are made true. A simple and efficient method for constraint resolution has been proposed in the work of Bjorn N. Freeman-Benson, John Maloney, and Alan Borning. We show how their method is related to the classical problem of graph matching, and from this connection we derive new resolution algorithms.

1 Introduction

A *constraint system* includes a set of variables and a set of relations among these variables, called *constraints*. The solution of a constraint system is an assignment of values to variables so that all, or many, of the relations are made true. In the perspective of computer programming, a constraint system lets us describe the value of a variable by the conditions on the variable, instead of the more imperative description which states the steps needed to compute the value.

The general question of constraint programming, even when taken in the delimited context of interactive graphics applications, requires a wide variety of combinatorial and numerical methods for solution. What methods are employed determine what types of problems can be solved.

The method of Bjorn N. Freeman-Benson, John Maloney, and Alan Borning, [Fre90] is of interest for its simplicity and efficiency. It directs the graph of variable interdependencies, interdependencies expressed as constraints, and deduces a straightforward sequence of function applications which will satisfy as many constraints as is possible without getting into complex equation solving, such as would be necessary if the graph had directed loops. We show how their method is related to the classical problem of graph matching, and from this connection we derive new resolution algorithms. These algorithms have been coded. We discuss the implementation details.

As in [Fre90], we emphasize in this report the incremental aspect of constraint solving, we are interested in methods which minimize the amount of work to be done when a new constraint or a new variable is added to or deleted from the system, or when a variable is bound to a new value.

2 The constraint resolution model

We adopt the following constraint satisfaction strategy. Assume that each constraint has associated with it various *methods* for satisfying the constraint. An n-ary relation has n methods, one for each variable in the constraint, each method expressing that variable as a function of the remaining $n-1$ variables. Application of any method associated with a constraint makes the constraint satisfied, because it calculates the proper value of the dependent variable. A dependent variable is said to be *determined* by the method that is applied in order to calculate its value. If for each variable we can select a method to determine it, if from each constraint one method is selected, and if no variable by composition of selected methods depends upon itself, we have a simple procedure for satisfying the constraints. List the methods such that no method has as an independent variable a variable determined by a method further down the list. Now, apply the methods in the order given by the list. More generally, we can have more constraints than variables, which implies that some constraints will have no method selected for them. After method selection, constraints are therefore of two types, *satisfied* or *unsatisfied*, depending on whether a method has or has not been selected for the constraint. A *resolution* is a list of methods verifying the condition above.

We can ensure that each variable is determined by some constraint by adding for each variable a constraint of arity one of the form either "let x:=x" or "let x:=constant". Such a constraint is always ready to determine the value of its variable if no other constraint has managed to do so. Constraints of this type are called *stay constraints* in [Fre90]. When a new variable is added to the constraint system, a stay constraint defining the initial binding for this variable is also added to the system. We also have *follow constraints* which bind variables to externally derived values. For example, when the user of a graphics editor drags a point to a new position in a constrained figure, a follow constraint binding the point coordinates to their new values is added to the constraint system.

The decision of which constraints to satisfy when conflicts arise is directed by the important notion of constraint hierarchies. The *locally-predicate-better* ordering [Fre90] is a partial order depending on an additional tag for each constraint, its *strength*. For each constraint c let $s(c) \in [0, r]$ be its strength, with r the strongest. The stay constraints are of strength 0, the follow constraints are of strength r. If A is a set of constraints, let A_i be the subset of A of constraints of strength i. We say that A is locally-predicate-better greater than B, l.p.b.-greater for short, if and only if there exists an integer i such that $A_j \supseteq B_j$ for all $j > i$ and $A_i \supset B_i$, strictly at strength i. To correspond exactly to the criteria used in [Fre90], a resolution must contain all constraints with highest strength to be considered a resolution at all. We shall not require this because we shall be concerned with l.p.b.-maximal resolutions and these resolutions will satisfy all highest strength constraints whenever this is possible. For a more general discussion of constraint hierarchies see [Bor88].

3 Constraint resolution and graph matching

Given a constraint system, we construct a bipartite graph G. The vertex bipartition of G is V, the set of variables used by the constraint system, and C, the set of constraints in the system. An edge (v, c) exists between $v \in V$ and $c \in C$ exactly when variable v is involved in constraint c. This graph is called the *constraint matching graph* and denoted $G_{cm}(V, C)$. A graph matching in $G_{cm}(V, C)$ is a a subset of the edges such that no two edges in the subset share a vertex. We say that a graph is matched if it has an associated matching. A matching in $G_{cm}(V, C)$ corresponds to a constraint resolution iff each vertex in V is incident to exactly one edge in the matching. An edge (v, c) in the matching means that from constraint c the method determining variable v has been selected. Conversely, a resolution translates to a matching in $G_{cm}(V, C)$, in fact, one which covers the set V.

We want a matching maximal in the l.p.b. ordering, called l.p.b.-maximal. To compare two matchings by l.p.b. compare the two sets of constraints covered by the matchings, that is, use the following definition:

$$M > N \iff \{\, c \,|\, (c, v) \in M \,\} > \{\, c \,|\, (c, v) \in N \,\}$$

The classical problem of graph matching addresses the problem of finding a *maximum matching*, a matching whose size is maximum. On the surface, these problems seem different — in short, do all l.p.b.-maximal resolutions have the same size?

Theorem 3.1 *Let $G = G_{cm}(V, C)$ be a constraint matching graph, M a matching in G. Denote by G_i the subgraph of G which omits all vertices from C of strength less than i, and M_i the matching in G_i induced by M. Then M is l.p.b.-maximal if and only if M_i is a maximum matching in G_i for all i.*

This gives immediately:

Corollary 3.1 *Given two matchings M and N in $G_{cm}(V,C)$ with M l.p.b.-maximal, let M_i and N_i be as above. Then N is l.p.b.-maximal if and only if $|M_i| = |N_i|$ for all i.*

To prove the theorem we return to a fundamental result in the theory of graph matching [Ber85]. Recall that a *path* in a graph is a sequence of edges from the graph with consecutive edges in the sequence sharing an endpoint, and is *simple* if no vertex is repeated in the path. An *alternating path* in a matched graph is a simple path such that the edges in the path alternate being and not being in the matching as one walks along the path. Recall also that a vertex in a matched graph is called *unsaturated* if it is not the endpoint of any edge in the matching. Hence, in a matched graph we can look for alternating paths which begin and end at unsaturated vertices. These are called *augmenting paths* because by "inverting the matching along the path", that is, taking the matched edges on the path out of the matching and putting the unmatched edges on the path into the matching, we end up with a new and larger matching. Note that a single edge which begins and ends at unsaturated vertices can be augmented.

Theorem 3.2 (Berge) *A matching in a graph is a maximum matching if and only if there are no augmenting paths.*

This theorem and the idea of inverting along an augmenting path, taken together, define an algorithm for finding maximum matchings. We return to this later, in particular, we give an algorithm for finding an l.p.b.-maximal matching. First, we prove Theorem 1 using Berge's theorem.

Proof: Suppose the induced matchings M_i are, for all i, maximum in graph G_i. If there exists a better matching N, that is, $N > M$, then by the definition of l.p.b. there exists an i for which the constraint vertices covered by N_i is a strict superset of the constraint vertices covered by M_i. In particular, $|N_i| > |M_i|$ and, both N_i and M_i being matchings in G_i, the maximum cardinality of M_i is contradicted.

Conversely, suppose for some i that M_i is not a maximum matching in G_i. By Berge's theorem there exists an augmenting path in G_i, which, because the graph is bipartite, has as one endpoint a vertex c from the set C. In G, this is an alternating path beginning at unsatisfied constraint c and ending at a variable either without matched constraint or matched to a constraint of strength less than that of c. By inverting along this path and new matching N is obtained with $N > M$. So M is not l.p.b.-maximal. □

This answers our initial question, l.p.b.-maximal resolutions satisfy the same number of constraints at each level of strength.

3.1 Static algorithm

Berge's theorem and the principle of inverting along an augmenting path form the basis of an algorithm which finds maximum matchings. Pick an unsaturated vertex, search the graph for an augmenting path beginning at this vertex, if such a path is found, invert along it. A vertex need only be considered once as the starting point for an augmenting path. If it becomes saturated, it stays saturated. If it cannot be saturated at the moment attention is focused on it, it will not be saturated later on. Interpreting saturation of a vertex in the set C as the satisfaction of a constraint, this process has the important and useful property that once a constraint becomes satisfied it stays satisfied during the search for other satisfiable constraints.

The following algorithm builds an l.p.b.-maximal matching M:

Algorithm 1 *Given a constraint matching graph $G_{cm}(V, C)$ and the empty matching $M = \emptyset$, take the constraints in C in nonincreasing strength and for each constraint c do: if c is satisfied go on to the next c. If c is unsatisfied, try to find an augmenting path starting from c. If found, invert it and go on to the next c. If there is no such path, go on to the next c. Exit immediately when $|M| = |V|$.*

The correctness of the algorithm is argued as follows. For any strength i, consider the situation after processing the last constraint of strength i. The set of saturated constraints will contain only constraints of strength i or greater. Hence this is a matching N in G_i. The algorithm would have proceeded exactly the same way if $G_{cm}(V, C)$ was replaced by G_i, so N is a maximum matching in G_i. From then on as the algorithm continues no satisfied constraint of strength i or greater become unsatisfied and so, at termination, M_i is of size at least $|N|$. As N is of maximum size, M_i is also. By Theorem 1, then, M is l.p.b.-maximal. To give the efficiency of the algorithm we need to specify how to search for an augmenting path. We construct a tree of alternating paths with root the given vertex by breadth first search. Each level of the tree alternates the search strategy. At the root and every second level afterwards, search all neighbors not yet examined, that is, vertices which are not yet in the tree. At the children of the root and every second level afterwards, search only the matched neighbor, there is at most one such neighbor. An augmenting path exists only if this tree has an unsaturated leaf. The breadth first search takes time order the number of edges, $O(|V||C|)$, and $O(|C|)$ searches are commissioned. Hence, the time complexity of the algorithm is $O(|V||C|^2)$.

3.2 Dynamic algorithm

Suppose we have a constraint matching graph and an l.p.b.-maximal matching. How quickly can we update the matching to maintain its maximality given atomic modifications to the constraint system? As in [Fre90], the five operations of interest are: add a variable, delete an existing variable, add a new constraint, delete an existing constraint, change the strength of an existing constraint. The first two operations, adding and deleting a variable, are simple. When adding a variable add also a stay constraint and match the variable with the constraint. When deleting a variable, no constraint but a single stay constraint should use that variable and consequently the variable and stay constraint are matched. Remove both the variable and the stay constraint. In both cases the resulting matchings are l.p.b.-maximal.

To add a constraint we apply the following algorithm. Assume that all the variables used by the constraint are already in the constraint matching graph and that, thanks to stay constraints, variables are always matched, that is, $|M| = |V|$. We assume further that this matching, the matching the algorithm starts with, is l.p.b.-maximal.

Algorithm 2 *Add the new constraint c and its incident edges to the graph. From among all constraints reachable from c by an alternating path of matched and unmatched edges choose one, say c', of minimum strength. If the strength of c' is strictly less than that of c, invert along the alternating path from c to c'. Otherwise, the matching is left unchanged.*

Theorem 3.3 *After running this algorithm, the invariants are restored in the resulting graph. That is, the matching is l.p.b.-maximal and all variables are matched.*

Proof: It is evident that whether or not an inversion occurs, the variables stay all matched. To prove the algorithm correct, we first establish a lemma.

Let G be the original graph, M its l.p.b.-maximal matching, and c the new constraint. Denote by $(G+c)$ the graph G with constraint c and its incident edges added. As before, G_i is the graph G restricted to constraints of strength i or greater, M_i the matching in G_i induced by M. Likewise, $(G+c)_i$ is the graph $(G+c)$ restricted to constraints of strength i or greater. Denote by $s(c)$ the strength of constraint c. Finally, denote by N an l.p.b.-maximal matching in $(G+c)$ and N_i the matching in $(G+c)_i$ induced by N.

Lemma 3.1 *Let $K = s(c)$ if M is l.p.b.-maximal in $(G+c)$, or if not*

$$K = \min\{\, i \mid M_i \textit{ not maximum in } (G+c)_i \,\} - 1.$$

Then

$$\begin{array}{lrcll} 1. & |N_i| & = & |M_i| & i > s(c) \\ 2. & |N_i| & = & |M_i| + 1 & s(c) \geq i > K \\ 3. & |N_i| & = & |M_i| & K \geq i \end{array}$$

Proof: Since for all $i > s(c)$, the graphs $(G+c)_i$ and G_i are the same, equation 1 is immediate.

Let i be a strength level such that M_i is not maximum in $(G+c)_i$. By Berge's theorem there is an augmenting path taking M_i to a new matching M_i'.

The action of inverting along the augmenting path introduces one new saturated constraint vertices and unsaturates none. However, since M_i' restricted to G_i cannot have more saturated constraint vertices than M_i, by the assumption that M_i is maximum in G_i, we know precisely the new saturated vertex in M_i': it is c. That is,

$$M_i'^{\circ} = M_i{}^{\circ} \cup \{c\},$$

where X° denotes the set of saturated constraint vertices in matching X. From this one easily deduces that,

$$M_j'^{\circ} = M_j{}^{\circ} \cup \{c\}, \quad s(c) \geq j > i.$$

Therefore,

$$|N_j| \geq |M_j'| = |M_j| + 1 \geq |N_j|, \quad s(c) \geq j \geq i,$$

where the first inequality follows from N being l.p.b.-maximal in $(G+c)$ and the last inequality expresses the fact that adding one new vertex to a graph increases the size of a maximum matching by at most one. Choosing i to be $K-1$, equation 2 follows.

For $K \geq i$, since M_i is maximum, the cardinality of N_i must equal it. This gives equation 3. □□

Suppose that M is not l.p.b.-maximal. Thus for i in the range $s(c) \geq i > K$ there are augmenting paths in $(G+c)_i$ and for other i not. These augmenting paths have one end on c, as has already been observed. So each augmenting path in $(G+c)_i$ gives an alternating path in $(G+c)$ from c to a constraint vertex c' with $s(c') < i$. Hence, the minimum strength of any constraint reachable by an alternating path starting from c, including c itself, is K. Inverting along the alternating path from c to c' of strength K gives a matching M' with the correct suite of cardinalities $|N_i|$, so it is l.p.b.-maximal.

Conversely, if M is l.p.b.-maximal, no such alternating paths can exist. □

To find the alternating path from c to a constraint of minimum strength, use breath first search starting from c. The exploration rules for the search are: if an unsatisfied constraint, which is only true for the root c, explore all neighbors;

if a satisfied constraint, explore all unexplored neighbors; if a variable, then explore its matched constraint. The form of the tree will be alternating levels of constraints and variables, the root an unsatisfied constraints, the internal nodes either variables or satisfied constraints, and all leaves satisfied constraints. All paths from the root are alternating.

To delete a constraint we apply to following algorithm. The assumptions are as before. In addition, we assume that the stay constraint will not be deleted if any other constraint has the constrained variable in its variable set.

Algorithm 3 *If the constraint to delete c is unsatisfied, delete it and all its incident edges and return the original matching M. Else, consider the set C of unsatisfied constraints reachable from c by simple alternating paths. If C is empty, c is the stay constraint for variable v, and v is has no other incident edges besides that from c. Delete v and c from the graph and return matching $M - (v, c)$. Otherwise, choose a constraint c' from C as strong as any in C, and invert along the alternating path from c to c' to form matching M'. Delete c and all its incident edges from the graph and return matching M'.*

Theorem 3.4 *After running this algorithm, the invariants are restored in the resulting graph. That is, the matching is l.p.b.-maximal and all variables are matched.*

Proof: The proof is similar to the constraint insertion case. The original graph and its restrictions are G and G_i. Deleting vertex c gives graphs $(G - c)$ and $(G - c)_i$. Let M be a l.p.b.-maximal matching in G, and N an l.p.b.-maximal matching in $(G - c)$.

Lemma 3.2 *Let $K = s(c)$ if c is unsatisfied in M. Otherwise let v be the variable matched to c and:*

$$K = \max\{\, i \mid M_i - (v, c) \text{ is not maximum in } (G - c)_i \,\}.$$

Then,

$$\begin{array}{rrcll} 1. & |N_i| & = & |M_i| & i > s(c) \\ 2. & |N_i| & = & |M_i| - 1 & s(c) \geq i > K \\ 3. & |N_i| & = & |M_i| & K \geq i \end{array}$$

Proof: Deleting a vertex can cause a maximum matching to loose at most one element, and can never cause the maximum matching to increase, i.e. $0 \leq |M_i| - |N_i| \leq 1$. If constraint c does not appear in M, then M is a matching in $(G-c)$ which makes this gap everywhere zero, hence M must be maximal. Choosing $K = s(c)$ makes equations 1 through 3 true in this case. On the other hand, suppose c appears in M, as edge (v, c). Since G_i and $(G-c)_i$ are the same for $i > s(c)$, equation 1 is true. Suppose $i \leq s(c)$. If $M_i - (v, c)$ is maximum in $(G-c)_i$ then $|N_i| = |M_i| - 1$. If it is not then, first of all, $|N_i| > |M_i| - 1$, therefore $|N_i| = |M_i|$, and second of all, we claim that for all $j < i$, $M_j - (v, c)$ is not maximum in $(G-c)_j$. For the augmenting path in $(G-c)_i$ must have one end on v, or else be present in G_i. The other end must be an unsatisfied constraint. The additional edges as one proceeds from $(G-c)_i$ to $(G-c)_j$ do not disquality this path as augmenting, so this path exists in $(G-c)_j$ as well. Equations 2 and 3 follow. □□

If c is not satisfied, the lemma shows that the suite of cardinalities of M is correct for it to be l.p.b.-maximal. In the case where c is matched with v and the only incident edge on v is from c, any augmenting path in the graph after removing c and v would exist in G, contradicting the l.p.b.-maximality of M. In the last case, letting $L = s(c')$, where c' is the constraint selected by the algorithm, it is clear that $L \leq s(c)$, else M would not be l.p.b.-maximal, and:

$$\begin{array}{llll} 1. & |M'_i| & = & |M_i| & i > s(c) \\ 2. & |M'_i| & = & |M_i| - 1 & s(c) \geq i > L \\ 3. & |M'_i| & = & |M_i| & L \geq i. \end{array}$$

We have but to show $L = K$, K being as defined in the lemma.

If $M - (v, c)$ is not maximal in $(G-c)_j$, there exists an augmenting path in $(G-c)_j$ beginning at v, if it did not begin at v this path would exists in G_j with matching M, contradicting our supposition on M. Therefore, $L \geq K$. Conversely, an augmenting path from v would imply that $M - (v, c)$ is not maximal in $(G-c)_j$, so $K \geq L$. □

To find the unsatisfied constraint of maximum strength and a simple alternating path leading to it, we again use breath first search starting from c and the exploration rules: if a variable, explore all unexplored neighbors; if a satisfied constraint, explore the matched variable; if an unsatisfied constraint, stop. The form of the tree will be alternating levels of variables and constraints, all internal nodes are satisfied constraints or variables, all leaves are either satisfied or unsatisfied constraints, and all paths from the root alternate.

Both the constraint insertion and deletion algorithms run in time proportional to the construction of the search tree, or $O(e)$, with e the number of edges in $G_{cm}(V, C)$. Already, $e \leq |V||C|$, but in practice the constraints will have bounded degree less than d, hence $e \leq d|C|$. In the case of insertion, only satisfied constraints will be searched, hence effectively $e \leq d|V| \leq d|C|$.

To modify the strength of a constraint we use the following algorithm.

Algorithm 4 *Let c be a constraint whose strength is changed from i to j. If $i < j$ and c is satisfied, exit. If $i < j$ but c is not satisfied, continue as if c of strength j was being added: if the satisfied constraint of minimum strength found by that algorithm is of strength less than j, invert the path. If $i > j$ and c is not satisfied, exit. If $i > j$ and c is satisfied, continue as if c was being deleted: if the unsatisfied constraint of maximum strength found by that algorithm is of strength greater than j, invert the path. Otherwise exit.*

Proof: Increasing the strength of a satisfied constraint, or decreasing the strength of an unsatisfied constraint will not destroy l.p.b.-maximality. Consider the increasing case. The G_k which change have $i < k \leq j$, and is this range the matching size increases by one, which is the largest increase possible. Hence all M_k are maximum. The decreasing case is similar.

The other two cases can be thought of as compositions of a deletion followed by a re-insertion at the new strength.

□

4 Constraint dependencies and resolution strategies

The previous section covered in detail the connection between constraint resolution and graph matching. Once a selection of methods is decided upon, we have to put those methods into operation. In the simplest case, we can order the methods in a list so that no method has as independent variable a variable determined by a method further down on the list. By applying the methods in the order given by the list, values are assigned appropriately to each variable. To illustrate: first the satisfied follow constraints are applied, writing their values into variables $v_1, \ldots, v_i$; next the satisfied stay constraints apply their stay methods to $v_{i+1}, \ldots, v_j$; next a method calculates v_{j+1} from some subset of $\{ v_1, \ldots, v_j \}$; and so on.

In general, such a simple strategy is impossible. We have not, for instance, avoided the possibility that methods c_i and c_j are both present in the resolution, where c_i calculates v_i from v_j and c_j calculates v_j from v_i. A cycle of data dependency is formed. Our solution is to let these cycles occur, isolate and solve them by more advanced strategies. The cycles themselves can be ordered, or more exactly the subgraphs of $G_{cm}(V, C)$ containing at least one cycle, thus there is no difficulty resolving the interaction of variables between two cycles. In effect, a cycle is considered as a large constraint. This super-constraint can be seen as a relation between all variables in all constraints in the cycle, and determines simultaneously all variables determined by all methods in the cycle.

This section describes the algorithms which find the cycles, order them, and simultaneously solve for the variables determined by the cycle. We consider only the case where the constraints in the cycle are linear equations.

Given a constraint matching graph $G_{cm}(V, C)$ and a maximal matching M, we construct a directed graph G_{cd} called the *constraint dependency graph*. The vertices are the set of satisfied constraints, and there is an edge from c to c' when solving c' requires a variable computed by c. That is:

$$(c, c') \in G_{cd} \iff \exists v \in V : \big((v, c), (v, c') \in G_{cm}(V, C) \big) \text{ and } \big((v, c) \in M \big).$$

Recall that a directed graph without directed cycles is called *acyclic*, and directed acyclic graphs, DAG's, can be *topologically ordered.* A topological ordering is an ordering of the vertices of the graph consistent with the direction of the edges. That is, if a directed edge from c to c' appears in the graph, then $c < c'$ in the topological order.

A directed graph with cycles does not have a topological ordering. However, equivalence classes of vertices can be defined, with $c \equiv c'$ if and only if there are directed paths both from c to c' and c' to c. These equivalence classes are called the *strongly connected components* of the graph. They form a DAG, the *reduced dependency graph*, denoted G^*_{cd}. The edges of G^*_{cd} are:

$$(\overline{c_0}, \overline{c_1}) \in G^*_{cd} \iff \exists c_0 \in \overline{c_0},\ c_1 \in \overline{c_1} :\ (c_0, c_1) \in G_{cd}.$$

One the other hand, these equivalence classes are as small as possible given that they must form a DAG. In particular, an acyclic constraint dependency graph induces itself as its reduced dependency graph.

Algorithms exist for finding strongly connected components and topological orderings, [Meh84], and have time efficiency $O(n+e)$, where n is the number of vertices in the graph, e the number of edges. Here n is the number of satisfied constraints which is equal to the number of variables, $|V|$. One bound for e is $n^2 = |V|^2$. However, if we make the reasonable assumption that constraints will have degree bound by d_c and variables will have average degree bound by d_v, the number of edges will be less than $d_c d_v |V|$. That is, in practice we expect these algorithms to run in time linear in the number of variables.

We now have all the machinery necessary to describe precisely how the constraint solver operates. The constraint matching graph is constructed and a matching selected. From this a constraint dependency graph is derived. The strongly connected components of the constraint dependency graph are calculated, and the reduced dependency graph derived. The reduced dependency graph is given a topological order. Apply each node in the reduced dependency graph in topological ascending order. What this means is the following: if the node represents a component consisting of a single method, simply apply the method; if the node represents a component consisting of many methods, say

$c_1, \ldots, c_r$, where c_i depends on the set of variables V_i and calculates v_i, then present to a special solver the data $C^o = \{ c_1, \ldots, c_r \}$, $V^o = \{ v_1, \ldots, v_r \}$ and $V^{in} = \cup V_i - V^o$. The solver is responsible for computing values for all variables in V^o so as to satisfy the constraints in C^o, using the values of the variables in V^{in} as read-only inputs. Note that the incremental algorithms described in the previous section cover only the first step. After each atomic modification of the constraint system, we know how to incrementally update the constraint matching graph and its associated matching. It is also straightforward to incrementalize the second step. However, we do not know of an algorithm to incrementally update the topologically ordered reduced dependency graph.

How the solver will work depends on the type of constraints in the strongly connected component. We cover the simplest case where each constraint is a linear equation among its variables. In effect, each method is a linear equation among its variables with a single variable alone on the right hand side of the equation. Let V_i be the variables in the method c_i and $v_i \in V_i$ be the determined variable. For m such methods in the strongly connected component, namely $c_1, \ldots, c_m$, the v_i are all unique. We can write:

$$A\overline{v} + B\overline{w} + \overline{c} = D\overline{v}$$

where $\overline{v} = (v_1, \ldots, v_m)^T$, $\overline{w}$ is a column vector of the other variables appearing in the c_i, $\overline{c}$ is a column vector of constants, and the matrices A, B and D are matrices formed from the coefficients of the c_i. Hence:

$$(D - A)\overline{v} = B\overline{w} + \overline{c}$$

and $(D - A)$ is m by m. If it is also invertible the super-method to apply is:

$$\overline{v} = (D - A)^{-1}(B\overline{w} + \overline{c}).$$

If it is not invertible, then the theory of the Moore-Penrose inverse [Cam79] can be applied.

We can avoid recomputing variables whose values will not change if we agree on a certain usage of the follow constraints. To some of the variables, those we consider as inputs, we apply follow constraints. These constraints are of absolute greatest strength and therefore will be satisfied — assuming at most one follow constraint incident on a given variable. The remaining variables will either be outputs of a method or outputs of the stay constraints. If only the values of the variables attached to follow constraints are modified, the entire list of methods need not be rerun. Only those methods which are successors of at least one follow constraint in G^*_{cd} have to be executed. We can lift out of the total order of G^*_{cd} a suborder including only these constraints. Thus, we recompute only what is necessary. A simple $O(n+e)$ algorithm for this is offered:

Algorithm 5 *Start with an empty list L^o and a list L of constraints in G^*_{cd} in topological order. Scan down L and for each $c \in L$ do: if c is a follow constraint or marked add it to the end of L^o and mark all direct successors of c in G^*_{cd}. At termination, L^o is a sublist of L containing all and only the successors of follow constraints. Because L^o is a sublist of L, that is, elements appear in the same order, L^o indicates a topological order on the subgraph of G^*_{cd} induced by taking only the constraints found in L^o.*

In the multiprocessing environment we can achieve greater efficiency by running solvers in parallel. Again using the topological order, fork the solver for a strongly connected component or the procedure associated to one method as a separate thread of control, one by one according to this order. Before launching the thread lock all its output variables. That is, if it is a method, lock its sole output variable, if it is a solver, lock all the output variables of all the methods sent to the solver. Each thread blocks until its inputs are unlocked. In this way computation evolves simultaneously among all threads and with the proper relationships.

5 Implementation

The graph matching and topological ordering algorithms have been implemented in Modula-3, an object oriented language which supports multithreading. An interface was constructed to allow a client to define and remove constraints and variables, to bind variables using follow constraints, to modify strengths, and to trigger a resolution.

The client creates variables. The constraint system knows about variables in so far as they are mentioned in a constraint. When the client wants to add a constraint, it calls the constraint system. The constraint system will then create references to the client's variables if the constraint uses them and no links already exist. A reference to the newly created constraint is returned to the client. In short, for a variable: the client creates it and the system points to it, for a constraint: the system creates it and the client points to it. However, constraints are opaque objects for the client.

When a variable is internalized by the constraint system, a stay constraint is created and attached to it. Follow constraints are created and removed by the "bind" and "unbind" commands, respectively, with argument the variable to be bound. As with stay constraints, the client cannot access the follow constraints. This is a difference between follow constraints and arity one constraints of maximum strength. Another difference is that the system remembers the currently bound variables. As explained in the text, it is possible to ask the system to update only variables which are successors of the bound variables. Hence the bound variables can be considered updatable. Targets of arity one constraints cannot.

There are two essential data structures inside the constraint system. The matching graphs and the dependency graph. The matching graph is represented by linked lists. The dependency graph uses arrays dimensioned to the number of vertices and vertices given integer names which index the array entries. This is done for speed. When the dependency graph changes the associated data structure is rebuilt from scratch.

Finally, the data flow structure keeps a list of the constraints in topological order, grouped into strongly connected components. Inside this list is indicated the sublist of constraints dependent on bound variables.

6 Conclusions and future directions

We have demonstrated a strong connection between constraint programming as described in [Fre90] and graph matching. This connection resulted in a theorem characterizing optimal resolutions of the constraint system and maximum matchings in its corresponding constraint matching graph. The maximum matching algorithm was extended to solve the constraint programming problem and its efficiency deduced.

Our approach differs from that of [Fre90] in two aspects. In our diagrams, both variables and constraints are represented by graph nodes. In [Fre90], constraints are instead represented by multi-tailed arrows. Also, the algorithm of [Fre90] introduces the notion of *walk-about strengths* in order to expedite constraint system updates. Although walk-about strengths can be simply transplanted into the graph algorithms presented here, we have not rigorously analyzed the strengths and weaknesses of this interesting idea.

References

[Ber85] Claude Berge, *Graphs*, North-Holland Mathematical Library, Vol. 6, 1985.

[Bor88] Alan Borning, Michael Maher, Amy Martindale, Molly Wilson, *Constraint Hierarchies and Logic Programming*, Proceedings of the Sixth International Logic Programming Conference, Lisbon, Portugal, 1989, pages 149–164.
Also in: Tech. Rep. 88–11–10, Computer Science Department, University of Washington, Nov. 1988.

[Cam79] S. L. Campbell, C. D. Meyer, *Generalized Inverses of Linear Transformations*, Pitman, 1979.

[Fre88] Bjorn Freeman-Benson, John Maloney, *The DeltaBlue Algorithm: An Incremental Constraint Hierarchy Solver*, Proceedings of the Eighth An-

nual IEEE International Phoenix Conference of Computers and Communications, 1989.
Also in: Tech. Rep. 88–11–09, Computer Science Department, University of Washington, Nov. 1988.

[Fre90] Bjorn Freeman-Benson, John Maloney, Alan Borning, *An Incremental Constraint Solver*, Communications of the ACM, 33(1):54–63, January 1990.

[Meh84] Kurt Mehlhorn, *Data Structures and Algorithms 2: Graph Algorithms and NP-Completeness*, EATCS Monographs on Theoretical Computer Science, Springer-Verlag, 1984.

[Mod-3] Luca Cardelli, James Donahue, Lucille Glassman, Mick Jordan, Bill Kaslow, Greg Nelson, *Modula-3 Report (revised)*, System Research Center Research Report 52, Digital Equipment Corporation, 1989.

[Nel85] Greg Nelson, *Juno, a constraint-based graphics system*, SIGGRAPH 1985 Conference Proceedings, San Francisco, 1985.