Problem Solving in the DISCO Constraint Database System*

Peter Z. Revesz

Department of Computer Science and Engineering
University of Nebraska, Lincoln, NE 68588 USA
revesz@cse.unl.edu

Abstract. This paper describes general approaches to solving two classes of problems using the DISCO constraint database system. The first class of problems occurs when distinct values from a subset of the integers must be assigned to the variables of a constraint satisfaction problem. The second occurs when a group of items must be selected from a subset of the integers such that each of a set of constraints holds.

1 Introduction

DISCO [1] is a constraint database system with a unique combination of two features: (1) it can accept (integer and set) constraint databases as inputs and (2) it can find a (constraint) database representation of the whole fixpoint model of each program on any given input database. No other system can do both. Constraint logic programming systems with sets like CLPS [7], Conjuncto [4], ECLIPSE [3], and {log} [2] can do (1) but cannot do (2). Database systems like CORAL [9] and LDL [15] can do (2) but cannot do (1).

The unique combination of achieving both (1) and (2) in DISCO is based on a bottom-up evaluation with existential quantifier-elimination over set and integer variables [10, 11, 14]. This paper gives examples that demonstrate that this method can be practical.

We look at two broad classes of problems: first, constraint satisfaction problems in which all of n variables $x_1, \ldots, x_n$ must be assigned distinct values from a subset of the integers D, and second grouping problems in which we need to select a group of values from a subset of the integers D such that each of a set of constraints holds. We also consider various subcases of these problems.

The paper is organized as follows. First, we review the DISCO constraint database system in Section 2. We discuss all distinct values constraint satisfaction problems in Section 3, and grouping problems in Section 4, and give some conclusions and future work in Section 5.

* This work was supported in part by NSF grants IRI-9625055 and IRI-9632871. This work appears in *Proc. Workshop on Constraint Databases and Applications*, Springer-Verlag LNCS 1191, pp. 302-315, Delphi, Greece, January 1997.

2 The DISCO Query Language

The syntax of the query language of DISCO [1], denoted $Datalog^{<\mathbf{Z},\subseteq_{\mathbf{P}(\mathbf{Z})}}$, is that of traditional Datalog (Horn clauses without function symbols) where the bodies of rules can also contain a conjunction of integer or set order constraints. That is, each program is a finite set of rules of the form: $R_0 :\!-\, R_1, R_2, \ldots, R_l$. The expression R_0 (the rule *head*) must be an atomic formula of the form $p(v_1, \ldots, v_n)$, and the expressions $R_1, \ldots, R_l$ (the rule *body*) must be atomic formulas of one of the following forms:

1. $p(v_1, \ldots, v_n)$ where p is some predicate symbol.
2. $v\theta u$ where v and u are integer variables or constants and θ is a relational operator $=, \neq, <, \leq, >, \geq, <_g$ where g is any natural number. For each g the atomic constraint $v <_g u$ is used as shorthand for the expression $v + g < u$.
3. $V \subseteq U$ or $V = U$ where V and U are set variables or constants.
4. $c \in U$ or $c \notin U$ where c is an integer constant and U is a set variable or constant.

Atomic formulas of the form (2) above are called gap-order constraints and of the form (3-4) are called set order constraints. In this paper we will always use small case letters for integer variables and capital letters for set variables. Set variables always stand for a finite or infinite set of integers.

Let $\mathcal{M}$ be the set of all possible ground tuples over the integers and sets of integers. Let P be a $Datalog^{<\mathbf{Z},\subseteq_{\mathbf{P}(\mathbf{Z})}}$ program and d be a constraint database. Let $\mathcal{D}$ be the set of ground tuples implied by d. Let us define the function T_P from and into $\mathcal{M}$ as follows.

$T_P(\mathcal{D}) = \{t \in \mathcal{M}$: there is a rule $R_0 :\!-\, R_1, \ldots, R_k$ in P and an instantiation $/$ such that $R_0/ = t$, and $R_i/$ holds if R_i is a constraint and $R_i/ \in \mathcal{D}$ otherwise for each $1 \leq i \leq k$. $\}$

The fixpoint model of a DISCO program P with input database D can be defined as $D \cup T_P(D) \cup T_P(T_P(D)) \cup \ldots$. It is shown in [1] that for $Datalog^{<\mathbf{Z},\subseteq_{\mathbf{P}(\mathbf{Z})}}$ queries the fixpoint model coincides with the least model and that the data complexity of DISCO queries is DEXPTIME-complete. (That means that we can express with a fixed DISCO program and variable input database of size n any problem that can be computed on a deterministic Turing machine in $2^{p(n)}$ time where $p(n)$ is some polynomial function of n.)

DISCO uses a bottom-up set-at-a-time semi-naive evaluation based on quantifier elimination over the above set of constraints. This guarantees to find all solutions. This is different from the top-down evaluation that is provided in constraint logic programming systems with set constraints. DISCO also allows infinite subsets of the integers, which usually systems with set constraints do not allow. (See [1] for more discussion and comparison among systems with set constraints.)

There are no special built-in relations in DISCO. However, in many problems that we will see the *pick* relation will be important. In DISCO, we can define the

pick relation $pick(c, A, A')$ which will be true if and only if $c \in A$ and $A' \subseteq A \backslash \{c\}$ where c is any integer constant. We define the pick relation by expressing for each c in the active domain (the set of constants that occur in the input database) the following input database tuple:

$$pick(c, A, A') :\!\!- c \notin A', A' \subseteq A, c \in A.$$

The following is a simple example of using the pick relation. Suppose that a set of people S is invited to a party. Find all possible set of people who may attend the party given that the second person and the seventh person cannot come. The solution using the pick relation would be the following.

$$attend(S'') :\!\!- invited(S), pick(2, S, S'), pick(7, S', S'').$$

Assuming 2 and 7 are in S, the above defines the set of people attending the party to be $S'' \subseteq S \setminus \{2, 7\}$ which is what we need. The DISCO evaluation will yield *a single constraint tuple* with a conjunction of set order constraints that is equivalent to the above constraint. For example, if $invited(\{1, 2, 3, 4, 5, 6, 7, 8, 9\})$ describes the set of people invited, then DISCO first makes the proper substitutions which yields the constraint tuple:

$$attend(S'') :\!\!- S = \{1, 2, 3, 4, 5, 6, 7, 8, 9\}, 2 \notin S', S' \subseteq S, 2 \in S, 7 \notin S'', S'' \subseteq S, 7 \in S'.$$

Then DISCO simplifies the above using the set constraint solving method in [11] to:

$$attend(S'') :\!\!- S'' \subseteq \{1, 3, 4, 5, 6, 8, 9\}.$$

This is the correct solution because any subset of the people in $\{1, 3, 4, 5, 6, 8, 9\}$ may or may not attend the party. Note that the DISCO solution is much more efficient than simply enumerating the set of possible subsets of $S \setminus \{2, 7\}$ which may be needed if we do not allow constraint tuples in the EDB (input) and IDB (defined) relations.

3 All Distinct Values Constraint Satisfaction Problems

In this section we describe general approaches to solving a class of constraint satisfaction problems in which all of n variables $x_1, \ldots, x_n$ must be assigned distinct values from a subset of the integers D.

3.1 First Approach Using Inequality Constraints

The first general approach is to use the following skeleton query:

$$\begin{aligned} assign(x_1,\ldots,x_n) \text{ :— } & range(x_1),\ldots,range(x_n),\\ & neq(x_1,x_2),\ldots,neq(x_{n-1},x_n),\\ & values(x_1),\ldots,values(x_n). \end{aligned}$$

The idea is to use a relation with n variables. Each variable is declared using the *range* relation to be an element of D. The range relation is described by constraints, typically by an upper and a lower bound constraint if D is a subsequence of the integers. Then an inequality constraint is declared between each pair of variables x_i, x_j for $1 \leq i < j \leq n$. Finally, each variable x_i is declared to be an integer in D using the relation *values*. The values relation is described using regular relational database tuples. In this skeleton solution we left out the constraints that are specific to each problem.

The range relation together with the other constraints is used to test whether there is any solution and to derive a constraint solution. If we leave out the values line, then DISCO will return the constraint solution as a set of constraint tuples. The values relation is needed only if we want to have a ground solution, i.e. a solution without any variables.

Example 3.1 There are seven cars *a,b,c,d,e,f,g* parking next to each other in some unknown order on one side of a street. We know that a and g do not park next to each other, that e parks in front of a with at least three cars between them, c parks behind b with at least two cars between them, f parks ahead of g with at least four cars between them, and b and f park next to each other. Write a query that finds all possible orders in which the seven cars may be parking.

In DISCO we can solve the above problem using the following query:

$$\begin{aligned} park(a,b,c,d,e,f,g) \text{ :— } & gap(a,g), e <_3 a, b <_2 c, f <_4 g, adj(b,f),\\ & range(a), range(b), range(c), range(d),\\ & range(e), range(f), range(g),\\ & neq(a,b), neq(a,c), neq(a,d), neq(a,e), neq(a,f), neq(a,g),\\ & neq(b,c), neq(b,d), neq(b,e), neq(b,f), neq(b,g), neq(c,d),\\ & neq(c,e), neq(c,f), neq(c,g), neq(d,e), neq(d,f), neq(d,g),\\ & neq(e,f), neq(e,g), neq(f,g),\\ & pos(a), pos(b), pos(c), pos(d), pos(e), pos(f), pos(g). \end{aligned}$$

In the above rule, the skeleton query was used with the addition of the first line which expresses the constraints that are special to this problem. The car in front of all other cars will be said to park in place 1 and the car at the end will be said to to park in place 7. We need to express the relations *adj* and *gap* which will be true if and only if the two cars do or do not park next to each other, respectively. The EDB (input) constraint database will be as follows:

$succ(1, 2)$.
$\vdots$
$succ(6, 7)$.

$adj(x, y) :\!\!-\ succ(x, y)$.
$adj(x, y) :\!\!-\ succ(y, x)$.

$gap(x, y) :\!\!-\ x <_1 y$.
$gap(x, y) :\!\!-\ y <_1 x$.

$range(x) :\!\!-\ 0 < x, x < 8$.

$neq(x, y) :\!\!-\ x < y$.
$neq(x, y) :\!\!-\ y < x$.

$pos(1)$.
$\vdots$
$pos(7)$.

This problem was solved in less than 5 seconds on a SUN sparc 2 workstation. The solution was $park(a, b, c, d, e, f, g) :\!\!-\ a = 5, b = 3, c = 6, d = 4, e = 1, f = 2, g = 7$. □

Remark: Some database systems allow constraint checking but not constraint solving. An example is CORAL [9], which would accept the above query as syntactically correct. However, CORAL would give a system error message "comparing two non-numbers" when $gap(a, g)$ is calculated. One thing that could be done in CORAL is to put the *pos* line first instead of last. But that would be extremely slow. This is because CORAL would take the cross-product of the *pos* relation with itself seven times before starting to check the constraints. Before the constraint checking starts, there would be $7^7 = 823,543$ number of tuples. In contrast, DISCO at first calculates a selection condition by constraint solving. The selection constraint is applied as early as possible keeping the size of the intermediate relations small. Hence, even if the same problem can be expressed in DISCO and other systems that allow constraint checking but not constraint solving, the DISCO solution will be often faster.

A variation of the all distinct values constraint satisfaction problem occurs when all distinct values from several sets $A_1, \ldots, A_k$ have to be matched together.

Example 3.2 A computer science department consists of six professors: Anderson, Brown, Clark, Davis, Edwards, and Fisher. The professors' office numbers are 101, 102, 103, 104, 105, and 106. Each professor has a separate office. Each professor is teaching exactly one course this semester. The courses offered are artificial intelligence, databases, graphics, hardware, software, and theory. We know the following information.

1. Anderson's office is 102 and Fisher's is 105.

2. The professor who teaches hardware has office 104.
3. Clark teaches software and his office is not 102.
4. Brown's office is 103 and she teaches databases.
5. Davis teaches theory.

Find the office numbers and the subjects taught by each professor.

In the following DISCO solution, let's assume that $ai = 1, databases = 2, graphics = 3, hardware = 4, software = 5, theory = 6$. We also number the professors in alphabetical order and refer to offices by the last digit of their numbers. Let the variables xo and xt denote the office number and the topic for professor x. In this problem we use these 12 variables as arguments of the output relation.

$$
\begin{aligned}
profs(ao, at, \ldots, fo, ft) :\!\!- \; & ao = 2, bo = 3, bt = 2, neq(co, 2), ct = 5, dt = 6, fo = 5, \\
& pair(ao, at), \ldots, pair(eo, et), \\
& range(ao), \ldots, range(fo), \\
& range(at), \ldots, range(ft), \\
& neq(ao, bo), \ldots, neq(eo, fo), \\
& neq(at, bt), \ldots, neq(et, ft), \\
& values(ao), \ldots, values(fo), \\
& values(at), \ldots, values(ft). \\
range(x) \quad :\!\!- \; & 0 < x, x < 7. \\
pair(x, y) \quad :\!\!- \; & x \neq 4, y \neq 4. \\
pair(4, 4). & \\
\\
values(1). & \\
\vdots & \\
values(6). &
\end{aligned}
$$

The complete program also includes the *neq* relation whose definition we did not repeat here. The *pair* relation states that each professor is either in office 104 and teaches hardware or is in some other office and teaches something else. This is the expression of the second constraint. The other constraints are straightforward and are shown in the first line. There are four different solutions to this problem. The DISCO system found all four solutions in 33 seconds. □

3.2 Second Approach Using the Pick Relation

The first approach uses for n variables $n \times (n-1)/2$ inequality constraints and at least $2n$ range constraints. As n gets larger, that may be too many constraints to store and to solve.

Our second approach to the constraint satisfaction problem with all distinct values takes advantage of the compact representation provided by sets and uses the pick relation defined at the end of Section 2.

The main idea behind this approach is to represent for each subproblem the yet unassigned values as a set A. Each time we assign to a new variable some constant c, which must be present in A. After the assignment is made, c must be taken away from A. This representation only requires propagating the current value of A to all subproblems. In many cases this propagation would result in more efficient solutions. The skeleton solution would look like this:

$$assign(x_1, \ldots, x_n) :\!\!- domain(A), pick(x_1, A, A_1), \ldots, pick(x_n, A_{n-1}, A_n).$$

Note that the above requires only $O(n)$ instead of $O(n^2)$ constraints. We will illustrate this approach in the following example.

Example 3.3 In the well-known SEND+MORE=MONEY problem we need to give for each letter a different digit value from 0 to 9 such that the resulting numbers satisfy the equality constraint. The skeleton query for this problem can be written down as follows:

$$\begin{aligned} assign(s,e,n,d,m,o,r,y) :\!\!- \; & domain(A), pick(m, A, Am), pick(o, Am, Ao), \\ & pick(s, Ao, As), pick(e, As, Ae), pick(n, Ae, An), \\ & pick(r, An, Ar), pick(d, Ar, Ad), pick(y, Ad, Ay). \end{aligned}$$

In this problem there are some constraints that are specific to this problem. These constraints are among the variables in each column of the summation. In particular for each column with carry-in c_i, input digits d_1 and d_2, output digit d_3 and carry-out c_o we have the following constraint

$$column(c_i, d_1, d_2, d_3, c_o) :\!\!- c_i + d_1 + d_2 = d_3 + 10 \times c_o$$

For the SEND+MORE=MONEY problem we have the following summation with the carry-in and carry-out variables also shown.

```
 c1 c2 c3 c4
    S  E  N  D
    M  O  R  E
--------------
 M  O  N  E  Y
    c1 c2 c3 c4
```

From this it is easy to see that the SEND+MORE=MONEY problem can be expressed as the set of column constraints:

$$\begin{aligned} c1 &= m \\ c2 + s + m &= o + 10 \times c1 \\ c3 + e + o &= n + 10 \times c2 \\ c4 + n + r &= e + 10 \times c3 \\ d + e &= y + 10 \times c4 \end{aligned}$$

It is easy to see that the values of the three variables on the left hand side uniquely determine the values of the two variables on the right hand side, i.e., the functional dependency $c_i, d_1, d_2 \rightarrow d_3, c_o$ holds. Since c_i is either 0 or 1

and d_1 and d_2 must be digits from 0 to 9, there are only 200 possible columns in this problem. Since DISCO currently does not allow linear constraints, we represented the column relation as a set of 200 ground tuples. We also used $domain(\{0,1,2,3,4,5,6,7,8,9\})$ as an input relation.

Now to solve the problem, we need to constrain the variables as early as possible by the *column* relation. Implicit in this problem is that no number starts with a 0 digit, hence m which is the carry-out in the leftmost column must be 1. We also have to note that in the rightmost column the carry-in must be 0. Hence the refined query would be the following:

$$\begin{aligned} assign(s,e,n,d,m,o,r,y) :\!\!- \; & c1 = 1, m = 1, column(c2,s,m,o,c1), domain(A), \\ & pick(m,A,Am), pick(o,Am,Ao), pick(s,Ao,As), \\ & column(c3,e,o,n,c2), pick(e,As,Ae), \\ & pick(n,Ae,An), pick(r,An,Ar), \\ & column(c4,n,r,e,c3), column(0,d,e,y,r4), \\ & pick(d,Ar,Ad), pick(y,Ad,Ay). \end{aligned}$$

This query was evaluated in DISCO on a SUN sparc 2 workstation in 186 CPU seconds. A nice thing about DISCO is that it evaluates all solutions by using a semi-naive bottom-up evalution technique. Hence we can be sure that the only solution is $assign(9,5,6,7,1,0,8,2)$, which was returned by DISCO. □

Remark: A similar word puzzle was solved using linear rational constraints in the ECLIPSE constraint logic programming system [8].

3.3 A Recursive Solution

In many problems we have to order a set of items enumerated from 1 to n, such that the ordering satisfies a single relation $constraint(y_1,\ldots,y_k)$ that tells which k items can be acceptable subsequences. This is a more restrictive case of the general problem of assignment from D because here $D = \{1,\ldots,n\}$ and because we only need to consider k adjacent items at a time. This allows a recursive solution, which is convenient when we want to write a fixed query that can be evaluated on several different input databases where n varies. In these cases the skeleton solution would look like this:

$$\begin{aligned} h(A) & :\!\!- assign(x_1,\ldots,x_k,A,n), no\_vars(n). \\ assign(x_2,\ldots,x_{k+1},A',i') & :\!\!- assign(x_1,\ldots,x_k,A,i), constraint(x_2,\ldots,x_{k+1}), \\ & \quad succ(i,i'), pick(x_{k+1},A,A'). \\ assign(x_1,\ldots,x_k,A,k) & :\!\!- init(A), constraint(x_1,\ldots,x_k), \\ & \quad pick(x_1,A,A_1),\ldots,pick(x_k,A_{k-1},A_k). \end{aligned}$$

Here the relation $assign(y_1,\ldots,y_k,A,i)$ is true if and only if there is an ordering such that the first i items are $x_1,\ldots,x_i$ and $y_{k-p} = x_{i-p}$ for $0 \le p < k$ and $constraint(x_{j+1},\ldots,x_{j+k})$ is true for each $1 \le j \le (i-k)$.

Here the last rule defines the base case when $i = k$, and in that rule A must be the set of items to be ordered. The intuition is that the items in A are picked in some order. The second rule defines an extension of the ordering by one such that the extension satisfies the constraint. The topmost rule defines the goal state, which occurs when the ordering extends to all n items. The topmost rule also assumes that the number of items is given as an input database fact.

Example 3.4 For example, testing whether an undirected graph has a hamiltonian cycle can be done using the above skeleton algorithm. A hamiltonian cycle is a path that starts and ends with the same vertex and goes through each vertex exactly once. Here the constraint is simply the edge relation $edge(x_1, x_2)$ which is true if and only if there is an undirected edge between vertices x_1 and x_2. To solve this problem, we try to find a path always going from the last vertex seen to an unvisited vertex. We keep always in A the set of vertices not yet visited on the path from the start vertex to the last vertex seen.

$$hamiltonian\_cycle(n) \text{ :— } path(x_1, A, n), start(x_2), edge(x_1, x_2), no\_vars(n).$$

$$path(x_2, A', i') \text{ :— } path(x_1, A, i), edge(x_1, x_2), succ(i, i'), pick(x_2, A, A').$$

$$path(x_1, A', 1) \text{ :— } vertices(A), start(x_1), pick(x_1, A, A').$$

Let A be the set of vertices of the input graph, and let n be the number of vertices. Here the third rule is the initialization, saying that we start from some start vertex specified in an input relation *start*. The relation $path(x_1, A, i)$ will be true if and only if between the start vertex and vertex x_1 there is a path that traverses i vertices but not those vertices which are in A. The second rule says that if we can visit i vertices from the start vertex such that the last vertex visited is x_1 and there is an edge from vertex x_1 to x_2 and x_2 was not yet visited, then $path(x_2, A', i + 1)$ will be true where A' is a subset of the previsouly known unvisited vertices A and does not contain x_2. Here the successor relation is an input relation that expresses the successor relation from 1 to n. Hence it has only $n - 1$ tuples.

Note that this problem requires an assignment of a distinct integer number between 1 to n to each vertex. This is done in the *path* relation. Here each proof tree of the *hamiltonian_cycle* relation will contain n tuples for the *path* relation. These n tuples will define a one-to-one mapping between the vertices and the time order in which they will be visited.

We could run the hamiltonian cycle query on various undirected graphs with upto 30 nodes. The running times ranged from 2 CPU seconds for a graph with 5 vertices to 316 CPU seconds for a graph with 30 vertices on a SUN sparc 2 workstation. □

4 Grouping Problems

In another class of problems, we need to select a group of items from a subset of the integers D such that a set of constraints all hold. We can express the skeleton solution for this class of problems as follows.

$select(S)$:— $satisfy(k, S), last(k)$.
$satisfy(j, S)$:— $satisfy(i, S), succ(i, j), constraint(j, S)$.
$satisfy(0, S)$:— $S \subseteq D, domain(D)$.

In this skeleton program the EDB relations are $domain(D), succ(i, j), last(k)$, and $constraint(j, S)$, while the IDB relations are $satisfy(j, S)$ and $select(S)$. Here $satisfy(j, S)$ is true if and only if S is a subset of D that satisfies the conjunction of the first j constraints on S. The following example illustrates the use of this skeleton query.

Example 4.1 A department needs to select a team of students to participate in a programming contest. The students eligible to participate are Jenny, David, Pat, Mark, Tom. Lilly, and Bob. The selection must satisfy the following requirements. If Bob is selected, then David must be selected. If both David and Pat are selected, then Mark cannot be selected. If both Tom and Jenny are selected, then Bob cannot be selected. If Pat is selected, then either Tom or Lilly must be selected; but Tom and Lilly cannot both be selected. Either Jenny or Lilly must be selected, but Jenny and Lilly cannot be both selected. Find all possible teams that may be sent to the programming contest.

In this solution let the numbers $1, \ldots, 7$ be the id numbers of Jenny, David, Pat, Mark, Tom, Lilly, and Bob in order. A DISCO solution would use the above skeleton query with the following EDB database:

$constraint(1, S)$:— $7 \in S, 2 \in S$.
$constraint(1, S)$:— $7 \notin S$.

$constraint(2, S)$:— $2 \in S, 3 \in S, 4 \notin S$.
$constraint(2, S)$:— $2 \notin S$.
$constraint(2, S)$:— $3 \notin S$.

$constraint(3, S)$:— $5 \in S, 1 \in S, 7 \notin S$.
$constraint(3, S)$:— $5 \notin S$.
$constraint(3, S)$:— $1 \notin S$.

$constraint(4, S)$:— $3 \in S, 5 \in S, 6 \notin S$.
$constraint(4, S)$:— $3 \in S, 5 \notin S, 6 \in S$.
$constraint(4, S)$:— $3 \notin S$.

$constraint(5, S)$:— $1 \in S, 6 \notin S$.
$constraint(5, S)$:— $1 \notin S, 6 \in S$.

$domain(\{1,2,3,4,5,6,7\})$.

$last(5)$.

$succ(0,1)$.
$\vdots$
$succ(4,5)$.

In this solution each of the five sentences is translated to a five groups of constraints. For example, the first sentence is translated to the two constraints shown in the first group. From each group one of the constraints must be true. For example, for the first group this means that either both Bob and David is in S (first constraint in group 1), or Bob is not in S (second constraint in group 1). □

A second example of using the skeleton query is the solution given in [1] for the problem of testing the satisfiability of a propositional formula in conjunctive normal form. In fact, there the skeleton query is used twice, once for assigning the variables, and second in testing whether each clause is satisfied. The reader is encouraged to look at the reference for further details.

A third example from [11] occurs in the problem of finding in an inheritance hierarchy the tightest upper and lower bounds on the elements of each class. This variation is interesting because here we deal with not a chain but a tree. We have to ensure that the set of elements S in each class s satisfies all upper and lower bound constraints on each path from the root to a leaf on which s lies.

4.1 A Second Approach Using the Pick Relation

A second approach to the grouping problem is to use the *pick* relation that was discussed before. The skeleton solution would be the following:

$$
\begin{array}{ll}
select(S) & :\!\!- satisfy(S,C,k), last(k). \\
satisfy(S,C1,j) & :\!\!- satisfy(S,C,i), succ(i,j), pick(c,C,C1), constraint(c,S). \\
satisfy(S,C,0) & :\!\!- S \subseteq D, domain(D), constraints(C).
\end{array}
$$

This skeleton query puts the id numbers of all constraints into a set C, which is given as an EDB relation *constraints*. At each application of the second rule, an element c is picked from C and it is checked whether the cth constraint holds. Hence it is easy to see that at all times $satisfy(S,C,j)$ is true if and only if the set S is a subset of D and S satisfies j number of constraints (or constraint groups) from C.

4.2 Combining All Distinct Values and Grouping Problems

If we want to enforce a set of constraints on a single set, then the order of constraints chosen does not matter. However, that is not the case if we need to enforce a set of constraints on the ordering of a set of elements. Previously, we

could enforce only a single constraint on the ordering. Using the above idea, we can write the following skeleton query that combines the recursive approach to ordering and grouping:

$$
\begin{array}{ll}
h(A) & :\!\!- \; assign(x_1, \ldots, x_k, A, n, C, m), no\_vars(n), \\
 & \quad no\_constraints(m). \\
 & \\
assign(x_2, \ldots, x_{k+1}, A', i', C, j) & :\!\!- \; assign(x_1, \ldots, x_k, A, i, C, j), \\
 & \quad succ(i, i'), pick(x_{k+1}, A, A'). \\
 & \\
assign(x_2, \ldots, x_{k+1}, A', i', C', j') & :\!\!- \; assign(x_1, \ldots, x_k, A, i), \\
 & \quad succ(j, j'), pick(c, C, C'), \\
 & \quad constraint(c, x_2, \ldots, x_{k+1}), \\
 & \quad succ(i, i'), pick(x_{k+1}, A, A'). \\
 & \\
assign(x_1, \ldots, x_k, A, k, C, 0) & :\!\!- \; init(A), constraints(C), \\
 & \quad pick(x_1, A, A_1), \ldots, pick(x_k, A_{k-1}, A_k).
\end{array}
$$

The skeleton query combines ideas from both Section 3.3 and Section 4. We have like in all-distinct values constraint satisfaction problems a set A of values that have to be assigned to the variables, and we also have like in grouping problems a set of constraints C. We use these two sets in parallel decreasing A once we picked a value from it and assigned that value to some variable, and decrease C as we satisfy more and more elements of it.

More precisely, in this skeleton query, the second rule extends the assignment and decreases A but leaves C unchanged. The third rule extends the assignment and decreases A and also verifies that a constraint c which was not shown to be satisfied yet is satisfied. The goal relation h is reached only if the assignment can be extended to all n variables and all the constraints are satisfied.

Intuitively, when extending an assignment the third rule should be applied instead of the second rule whenever possible. This can be often enfored by adding extra constraints to the two rules. Without that the efficiency of the algorithm would be bad because the second rule would generate all possible assignments.

Example 4.2 An example of applying the above skeleton query occurs in genome mapping. In this problem we decide whether there is an ordering for n genes on a string of genes that satisfies each of m different adjacency constraints among the genes. In this case, it is easy to see that we have to apply rule three m times and rule two $d = n - m$ times. We create an EDB relation $limit(d)$ that contains the value of d. The solution can be expressed as follows.

$has\_seq(x_1)$:— $assign(x_1, A, n, C, m, d), no\_genes(n), no\_constraints(m).$

$assign(x_2, A1, i1, C, j, d1)$:— $assign(x_1, A, i, C, j, d), d > 0, succ(d1, d), succ(i, i1), pick(x_2, A, A1).$

$assign(x_2, A1, i1, C1, j1, d)$:— $assign(x_1, A, i, C, j, d), pick(x_2, A, A1), adjacent(s, x_1, x_2), pick(s, C, C1), succ(i, i1), succ(j, j1).$

$assign(x_1, A1, 1, C, 0, d)$:— $items(A), first(x_1), pick(x_1, A, A1), constraints(C), limit(d).$

For seven genes and six adjacency constraints, we could run in DISCO on a SUN sparc 2 workstation within 112 CPU seconds the above algorithm. The above algorithm can be also extended to cases where contiguity constraints are given among three or more genes. A contiguity constraint on k genes means that these k genes must form a subsequence of the entire genome. □

5 Conclusions and Future Work

In future versions of DISCO we plan to add more optimization methods. We also plan to add a limited form of negation that allows the computation of the whole perfect model of a program on a constraint database input [12]. This would allow more types of problems to be expressed in DISCO.

We also look for new problems. Admittedly, the SEND+MORE=MONEY problem is not a great argument for DISCO because linear constraints are quite natural here and because it has only one solution. When there is only one solution, then in general computing the solution by CLP systems will be faster. It is like a lake with a single fish. You could better catch that fish with a hook than with a net. We must look for lakes that teem with fish. Recently we investigated a genomic database application where the advantage of DISCO over other systems is clearer [13].

References

1. J. Byon, P. Z. Revesz. DISCO: A Constraint Database System with Sets. *Proc. Workshop on Constraint Databases and Applications*, Springer-Verlag, LNCS 1034, pp. 68–83, September 1995.
2. A. Dovier, G. Rossi. Embedding extensional finite sets in CLP. *International Logic Programming Symposium*, 1993.
3. ECLIPSE. *Eclipse user manual.* Technical report. ECRC, 1994.
4. C. Gervet. Conjunto: Constraint Logic Programming with Finite Set Domains. *Proc. International Logic Programming Symposium*, 339–358, 1994.
5. P. C. Kanellakis, G. M. Kuper, P. Z. Revesz. Constraint Query Languages. *Journal of Computer and System Sciences*, vol. 51, 26–52, 1995.
6. G. M. Kuper. Logic Programming with Sets. *Journal of Computer and System Sciences*, vol. 41, 44-64, 1990.

7. B. Legeard, E. Legros. Short overview of the CLPS System. *Proc. PLILP*, 1991.
8. D. Pountain. Constraint Logic Programming. *Byte*, February 1995.
9. R. Ramakrishnan, D. Srivastava, S. Sudarshan. CORAL: Control, Relations and Logic. *Proc. VLDB*, 1992.
10. P. Z. Revesz. A Closed Form Evaluation for Datalog Queries with Integer (Gap)-Order Constraints, *Theoretical Computer Science*, vol. 116, no. 1, 117-149, 1993.
11. P. Z. Revesz. Datalog Queries of Set Constraint Databases, *Fifth International Conference on Database Theory*, Springer-Verlag LNCS 893, pp. 425–438, Prague, Czech Republic, January, 1995.
12. P. Z. Revesz. Safe Stratified Datalog with Integer Order Programs, *First International Conference on Principles and Practice of Constraint Programming*, Springer-Verlag LNCS 976, pp. 154–169, Cassis, September, 1995.
13. P. Z. Revesz. Genomic Database Applications in DISCO. *CP96 Workshop on Constraints and Databases*, Cambridge, Massachusetts, August, 1996.
14. D. Srivastava, R. Ramakrishnan, P.Z. Revesz. Constraint Objects. *Proc. 2nd Workshop on Principles and Practice of Constraint Programming*, 274–284, 1994.
15. S. Tsur and C. Zaniolo. LDL: A Logic-Based Data-Language. *Proc. VLDB*, pp 33-41, 1986.

This article was processed using the LaTeX macro package with LLNCS style