Vol. 41 No. 6 SCIENCE IN CHINA (Series E) December 1998

OLDTNF-based evaluation method for handling recursive queries in deductive databases

CHEN Yangjun (陈阳军)

(Technical Institute of Changsha, Changsha 410004, China; Department of Computer Science, TU Chemnitz, 09107 Chemnitz, Germany)

Received August 28, 1997

Abstract OLDTNF resolution is an important mechanism used in a Prolog interpreter. This mechanism is extended and improved for evaluating recursive queries in deductive databases. The key idea of the refinement is to distinguish between two classes of lookup nodes in an OLDTNF derivation and to handle them differently. First, reduce the search space by cutting of any subtree rooted at a lookup node of the first class. Further, speed up the evaluation by processing the second class in a second phase and generate many solutions directly from the solutions already produced (and the corresponding keys of solution lists) instead of evaluating them by expanding the corresponding subtrees in terms of the new solutions stored in solution lists.

Keywords: Prolog, SLD resolution, OLDTNF resolution, deductive databases, recursion.

It has been recognized for some time that first-order database query languages are lacking in expressive power[1]. Since then many higher-order query languages have been investigated in the context of deductive databases. A language that has received considerable attention recently is Datalog, the language of logic programs (known also as Horn-clause programs) without function symbols, which is essentially a fragment of fixpoint logic[1]. A canonical example of Datalog is the following program that computes transitive closure, where we think of the database as a directed graph.

$$\text{path}(x, y) :- \text{edge}(x, y),$$
$$\text{path}(x, y) :- \text{edge}(x, z),\ \text{path}(z, y).$$

In this example, we take edge(·, ·) to be an extensional database (EDB) predicate, that is, representing basic facts stored in the database. For example edge(1, 5) is an EDB fact stating that there is an edge between nodes 1 and 5. The intensional database (IDB) predicate path(·, ·) represents facts deduced from the database via the logic program above: the first rule says that every directed edge forms a path, and the second rule tells how paths can be joined together. We can now query, for instance, path(1, 7) or path(2, v) to determine, respectively, whether there is a path from node 1 to node 7, or what nodes v are connected to node 2 by a path.

Recent works have addressed the problems of finding efficient evaluation methods for Datalog queries and developing optimization techniques for Datalog[2-6]. By "efficient", we refer to the size of the database. A typical approach to the problem of efficient evaluation involves identifying "nice" properties of Datalog programs that facilitate efficient computation of programs with these properties. For example, Ullman and Van Gelder[7] have identified the polynomial fringe property of Datalog programs, where every proof involves a number of EDB facts (at the "fringe" of the proof tree) at most polynomials in the size of the database. Although the complexity of evaluating arbitrary Datalog programs can be PTIME-complete[8], Ullman and Van Gelder have shown that the complexity of Datalog programs with the polynomial fringe property is in NC[7], that is, all facts can be deduced in parallel time polynomial in the logarithm of the size of the database, given a number of processors polynomial in the size of the database.

At first, the problem of optimizing Datalog queries does not seem to be too difficult, since every rule in a Datalog program can be viewed as a conjunctive query. Conjunctive queries constitute a fragment of the class of first-order queries for which the optimization problem is completely solved[9]. Unfortunately, it is the recursive application of the rules that makes Datalog queries hard to evaluate. A possibility of optimizing Datalog queries is to remove recursions and to transform the original program to an equivalent one that contains no recursion. However, deciding whether it is possible to have some or all of the recursive predicates of a program removed, is undecidable[10]. Then, we have to find an efficient method for a "direct" evaluation.

In this paper, we identify a new "nice" property, the solution similitude property, and try to extend OLDTNF resolution to a top-down but set-oriented method for evaluating recursive queries against a stratified database, thereby using the solution similitude property to improve the efficiency.

We say that a database is stratified if there exists a level mapping r, so that for every clause in the database (of the form):

$$B :- L_1, \cdots, L_n.$$

If L_i is positive, that is, an atom, then $r(L_i) \leqslant r(B)$, and if L_i is negative, then $r(L_i) < r(B)$, for all $i, 1 \leqslant i \leqslant n$. So, in a stratified database, recursion via negation as in the following program is not allowed.

$$p :- \neg q,$$
$$q :- p.$$

The main obstacles which have to be faced in a top-down evaluation of recursive queries against a stratified database are

- Floundering (due to negated body literals), and
- Infinite derivations (due to the presence of recursions).

We say that a query is floundering if during the evaluation of the query a negated body literal with unrestricted variable is encountered. It leads to the problems of domain-dependency[5] and inefficiency. Today, this problem is well understood. The simplest possibility to exclude floundering computations is to restrict the attention to allowed queries, which were introduced by Clark[11]. (A rule is Clark-allowed if every variable appears in at least one positive body literal). However, to avoid the infinite derivations, the subsumption has to be checked to cut off any infinite branch. The problem is that cutting off an infinite branch may affect the completeness and thus a mechanism has to be designed to find the lost solutions again. To this end, OLDTNF resolution[2,4] was proposed, which distinguishes between two kinds of nodes: solution nodes and lookup nodes, and handle them differently based on the tabulation technique. In this way, both termination and completeness can be guaranteed. We show that this method proceeds redundantly in certain cases and can be improved by further differentiating between two classes of lookup nodes. First, we try to reduce the search space that will be traversed by OLDTNF resolution. We do this by recognizing all similar portions of a graph and manage to produce all the relevant solutions by constructing only one of them (based on the identification of the first class of lookup nodes). The other refinement is concerned with the treatment of cyclic data. In this case, a cycle is stored when it is encountered at the first time (based on the identification of the second class of lookup nodes). We then suspend the traversal along the corresponding path to avoid duplicate work. However, as many intermediate solutions may not be used to produce new solutions along a cyclic path, suspending the traversal along the cyclic path may prevent us from obtaining all solutions. Therefore, we develop a process to evaluate the remaining solutions by iterating on each cyclic path with a different initial value each time. In this iteration process, we further optimize the evaluation by generating most solutions for cyclic data directly from the solutions already found and the associated path information instead of traversing the relevant subgraphs as usual. In this way, we can decrease the time complexity by an order of magnitude or more. This is because traversing paths requires accesses to the external storage or search of large relations but the "generating" operations happen always in main memory and require only accesses to small data sets (i. e. the solutions already found).

1 OLDTNF resolution

OLDTNF resolution is based on OLDT resolution, augmented with negation as failure rule, just as SLDNF resolution is based on SLD resolution[1]. OLDT resolution (OLD resolution with tabulation) is a mechanism used in a top-down Prolog interpreter with memo-ization. It was proposed by Tamaki-Sato[2], and similar work has been done by several researchers[3,12]. Recently, OLDT resolution is widely used as an abstract interpretation engine[13,14].

The basic principle of OLDT resolution is to prevent the interpreter from repeatedly trying to solve the same goal (in the case of recursive programs) and thus to cut off any infinite computation path, by introducing the tabulation technique into OLD derivation. (For the other top-down strategies, different techniques are used[3]). An informal explanation of the behavior of OLDT resolution is as follows. When a goal first appears in a computation path, it is said to be a solution node. All solutions obtained from the solution node are stored in a table called the solution table. On the other hand, when a similar goal appears in a computation path, it is said to be a lookup node. A lookup node is resolved only with the atoms in the solution table, i. e. atoms stored in the solution table are used as lemmas. When there is no solution resolvable with a lookup node, the computation of the lookup node suspends until new resolvable solutions are registered in the solution table.

Now we describe OLDTNF resolution according to ref. [4] except for the use of association, the use of variant of instance, and the definition of OLDTNF subrefutation.

First, we introduce the following important concepts.

A search tree is a tree satisfying the following conditions: (i) Each node is classified into either a solution node or a lookup node and is labeled with a pair of a (possibly empty) goal and a substitution. (The distinction between solution nodes and lookup nodes can be seen later). (ii) Each edge is labeled with a substitution.

A search tree of $G\sigma$ (G is a goal and σ is a substitution) is a search tree whose root node is labeled with (G, σ). A node in a search tree is called a null node when the goal part of the label is $\square$ (which represents the empty goal). When a node in a search tree is labeled with ("$A_1, A_2, \cdots, A_N$", σ), the literal $A_1\sigma$ is called the head literal of the node. A solution table is a set of entries. Each entry is a pair of the key and the solution list. The key is a literal such that no variants of this key appear (as keys) elsewhere in the solution table. The solution list is a list of literals, called solutions, such that each solution in it is an instance of the corresponding key. Let Tr be a search tree and Tb be a solution table. An association of Tr and Tb is a set of pointers connecting from each lookup node in Tr into some solution list in Tb such that the head literal of the lookup node and the key of the solution list are variants of each other. The tail of the solution list pointed from a lookup node is called the associated solution list of the lookup node.

An OLDTNF structure of $G\sigma$ is a triple (Tr, Tb, As), where Tr is a search tree of $G\sigma$, Tb is a solution table, and As is an association of Tr and Tb. A node in a search tree of the OLDT-NF structure (Tr, Tb, As) labeled with ("$A_1, A_2, \cdots, A_n$", σ) is said to be OLDTNF resolvable when it satisfies either of the following conditions: (i) The node is a leaf solution node of Tr, and there is some definite clause "B :- $B_1, B_2, \cdots, B_m$" ($m \geqslant 0$) in program P such that $A_1\sigma$ and B are unifiable, say by an m.g.u. (most general unifier) θ. (We assume that, whenever a clause is used, a fresh variant of the clause is used.) The pair of the (possibly empty) goal "$B_1, B_2, \cdots, B_m, A_2, \cdots, A_n$" and the substitution $\sigma\theta$ (or possibly the restriction of $\sigma\theta$ to the variables in "$B_1, B_2, \cdots, B_m, A_2, \cdots, A_n$") is called the OLDTNF resolvent. (ii) The node is a lookup node of Tr, and for some substitution θ (for the variables in $A\sigma$), there is a solution A in the associated solution list of the lookup node such that $A = A_1\sigma\theta$. The pair of the (possibly empty) goal "$A_2, \cdots, A_n$" and the substitution $\sigma\theta$ (or possibly the restriction of $\sigma\theta$ to the variables in "$A_2, \cdots, A_n$" is called the OLDTNF resolvent. (iii) The node is a negative node. Let $A_1 = \neg A_0$. That is, A_1 is a negated literal. We create a new initial OLDTNF structure of A_0, where its solution table is also newly created. If there is a finitely failed OLDTNF structure, then ("$A_2, \cdots, A_n$", σ) is called the OLDTNF resolvent.

An OLDTNF subrefutation of a literal and an OLDTNF subrefutation of a goal are paths in a search tree (not necessarily starting from the root node) which are simultaneously defined inductively as follows:

- A path with length more than 0 starting from a solution node is an OLDTNF subrefutation of an atom $A\sigma$ with solution $A\tau$ when
  - the initial node is labeled with a pair of the form ("A, G", σ), the initial edge with, say substitution θ, and the last node with a pair of the form ("G", σ');
  - the node next to the initial node is labeled with a pair of the form ("$A_1, A_2, \cdots, A_n, G$", $\sigma\theta$), and the path except the initial node and the initial edge is a substitution of $(A_1, A_2, \cdots, A_n)\theta$ with solution $(A_1, A_2, \cdots, A_n)\theta\tau'$ ($n \geqslant 0$); and
  - τ is $\sigma\theta\tau'$.
- A path with length 1 starting from a lookup node is an OLDTNF subrefutation of an atom $A\sigma$ with solution $A\tau$ when
  - the initial node is labeled with a pair of the form ("A, G", σ), the initial edge with, say substitution θ, and the last node with a pair of the form ("G", σ''); and
  - τ is $\sigma\theta$.
- A path with length 1 starting from a negative node is an OLDTNF subrefutation of a negated literal $\neg A_0$ with solution "true" and identity substitution when
  - the initial node is labeled with a pair of the form ("$\neg A_0, G$", σ) and the last node with a pair of the form ("G", σ).
- A path with length 0, i.e. a path consisting of only one node, is an OLDTNF subrefuta-

tion of $\Box\sigma$ with solution $\Box\sigma$.

- A path with length more than 0 is an OLDTNF subrefutation of a goal $(A_1, A_2, \cdots, A_n)\sigma$ with solution $(A_1, A_2, \cdots, A_n)\tau(n>0)$ when
  - the initial node is labeled with a pair of the form ("$A_1, A_2, \cdots, A_n, H$", σ), and the last node with a pair of the form ("H", σ');
  - the path is the concatenation of a subrefutation of $A_1\sigma$ with solution $A_1\sigma\tau_1$, a subrefutation of $A_2\sigma\tau_1$ with solution $A_2\sigma\tau_1\tau_2, \cdots$, a subrefutation of $A_n\sigma\tau_1\tau_2\cdots\tau_{n-1}$ with solution $A_n\lambda\tau_1\tau_2\cdots\tau_{n-1}\tau_n$; and
  - τ is $\sigma\tau_1\tau_2\cdots\tau_{n-1}\tau_n$.

In particular, a subrefutation of $A\sigma$ is called a unit subrefutation of $A\sigma$.

The initial OLDTNF structure of $G\sigma$ is the OLDTNF structure (Tr_0, Tb_0, As_0), where Tr_0 is a search tree consisting of only the root solution node labeled with (G, σ), Tb_0 is the solution table consisting of only one entry whose key is the head literal of the root node and whose solution list is an empty list [ ], and As_0 is an empty set of pointers.

An immediate extension of OLDTNF structure (Tr, Tb, As) in program P is the result of the following operations, when node v of OLDTNF structure (Tr, Tb, As) is OLDTNF resolvable.

- When v is a leaf solution node, let $C_1, C_2, \cdots, C_k(k\geqslant 1)$ be all the clauses with which the node v is OLDTNF resolvable, and $(G_1, \sigma_1), (G_2, \sigma_2), \cdots, (G_k, \sigma_k)$ be the respective OLDTNF resolvent. Then add k child nodes of v labeled with $(G_1, \sigma_1), (G_2, \sigma_2), \cdots, (G_k, \sigma_k)$ to v.
- When v is a lookup node, let $A\sigma\theta_1, A\sigma\theta_2, \cdots, A\sigma\theta_k(k\geqslant 1)$ be all (the variants of) the solutions in the associated solution list with which node v is OLDTNF resolvable, and $(G_1, \sigma_1), (G_2, \sigma_2), \cdots, (G_k, \sigma_k)$ be the respective OLDTNF resolvent. Then add k child nodes of v labeled with $(G_1, \sigma_1), (G_2, \sigma_2), \cdots, (G_k, \sigma_k)$ to v.
- When v is a negative node ("$A_1, A_2, \cdots, A_n$", σ), let A be the head literal of the node and $A = \neg A_0$. We create a new initial OLDTNF structure T_0 of A_0, where its solution table is also newly created. We consider two cases. If there exists an extension (see below) of T_0 which contains a success leaf, then v has no child and it is a failure leaf. If there is a finitely failed OLDTNF structure which is an extension of T_0, then v has a unique child node v' of the form ("$A_2, \cdots, A_n$", σ). The edge from v to v' is labeled with identity substitution.
- In all the above cases, the edges from v to the node labeled with (G_i, σ_i) is labeled with θ_i, where θ_i is the substitution of the OLDTNF resolution. A new node is a lookup node when the head literal is a variant of some key in Tb, and is a solution node or a negative node otherwise. If a new node is a lookup node, add a pointer from the new lookup node to the head of the solution list of the corresponding key. If a new node is a solution node, add a new entry whose key is the head atom of the new node and whose solution list is an empty list. Otherwise, it is a negative node.
- For each unit subrefutation of literal $A\sigma$ (if any) starting from a solution node and ending with some of the new nodes, add its solution $A\tau$ to the end of the solution list of $A\sigma$ in Tb, if $A\tau$ is not in the solution list.

An OLDTNF structure (Tr', Tb', As') is an extension of OLDTNF structure (Tr, Tb, As) *if* (Tr', Tb', As') is obtained from (Tr, Tb, As) through successive applications of immediate extensions. Note that an immediate extension is applicable to any lookup node (so long as its associated solution list is nonempty), whereas it is applicable to only leaf solution nodes and leaf negative node.

An OLDTNF refutation of $G\sigma$ in program P is a path from the root node to a null node in the search tree of some extension of the initial OLDTNF structure of $G\sigma$. The solution of an OLDTNF refutation is defined in the same way as that of an OLDTNF subrefutation.

2 Main ideas of refinements

In the following discussion, we confine ourselves to the allowed stratified databases. In addition, as with the other evaluation strategies, a preprocessor is implemented to reorder the body literals such that the literals with some of their variables bound to constants (in terms of the query submitted to the system) are before the literals whose variables have no bindings. Furthermore, any positive literal directly or indirectly correlated with a negative literal should be put before the negative literal. (We say that two literals are correlated if they have at least one shared variable.) In this way, any reordered allowed program will have the property that when a negative body literal is encountered during a top-down evaluation, all its arguments become instantiated. Using the negation-as-failure rule, such a negative body literal can always be evaluated in finite time. For the optimization purpose, an extra step is required. If any two correlated non-recursive literals are separated by a recursive predicate, we shift the latter such that both of them are before the recursive predicate. This requirement is not only for optimization, but also for the application of the technique for generating answers directly (see below). An example of so-reordered rules is the nonlinear version of the same-generation program (given a query like ?- sg(john, y))

sg (x, y) : - flat (x, y),

sg (x, y) : - up(x, z_1), sg (z_1, z_2), flat (z_2, z_3), sg (z_3, z_4), down (z_4, y).

Another example is the definition of reverse function (given a query like ?- reverse ($\{1,3,5,4,2\}, y$))

reverse $(M, [x \mid L])$: - append $(N, [x], M)$, reverse (N, L),

reverse $([\,], [\,])$,

append $([y \mid N], K, [y \mid M])$: - append (N, K, M),

append $([\,], K, K)$.

2.1 Subsumption checks

We begin the discussion with an exact definition on similar goals.

Definition 2.1. A non-negative node of the form: ("$A_1, A_2, \cdots, A_n$", σ_1) is subsumed by another non-negative node of the form: ("$B_1, B_2, \cdots, B_m$", σ_2) if A_1 and B_1 have the same predicate symbol and at the same time $A_1\sigma_1$ has the same bound arguments as $B_1\sigma_2$. For example, node $(s(z_2, w_2), q(w_2, y_2), q(y_2, y_1); \{z_2/a_1\})$ of the graph shown in fig. 1 is subsumed by $(s(a_1, y); \{\})$, because both of them have the same bound argument.

In addition, a node is usually thought of as being subsumed by itself.

Definition 2.2. A repeated incomplete lookup node (RILN) is a node which is subsumed by a previous node which has appeared earlier on the same path as the RILN. For example, node

$(s(z_2, w_2), q(w_2, y_2), q(y_2, y_1); \{z_2/a_1\})$ of the graph shown in fig. 1 is an RILN because it is subsumed by $(s(a_1, y); \{\})$ which has appeared earlier on the same path.

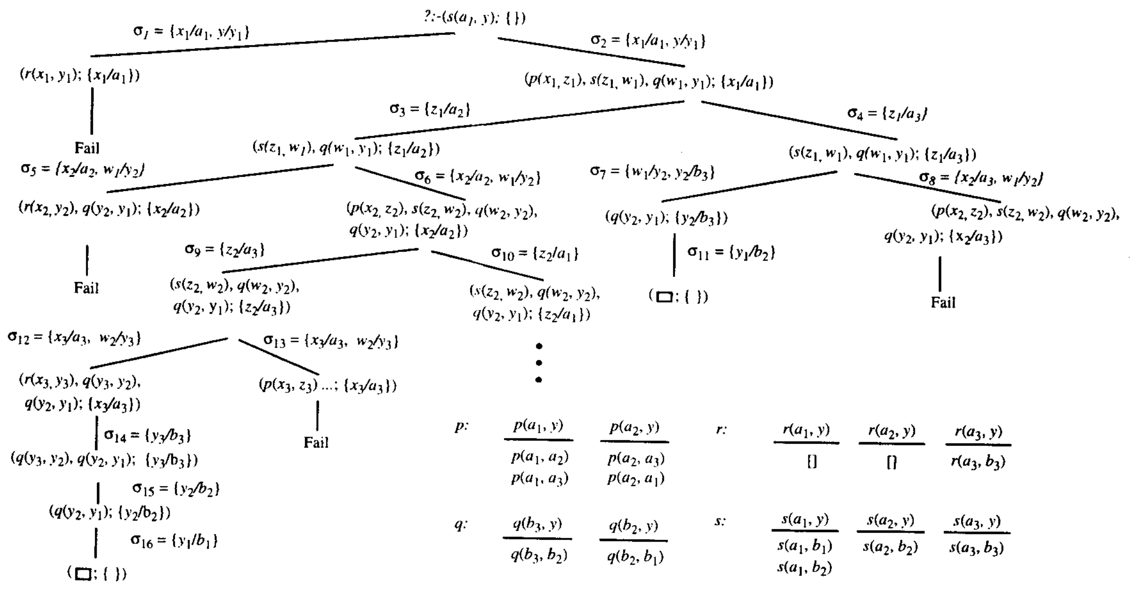

?:-(s(a_1, y); { })
σ_1 = {x_1/a_1, y/y_1}
σ_2 = {x_1/a_1, y/y_1}
(r(x_1, y_1); {x_1/a_1})
(p(x_1, z_1), s(z_1, w_1), q(w_1, y_1); {x_1/a_1})
σ_3 = {z_1/a_2}
σ_4 = {z_1/a_3}
Fail
(s(z_1, w_1), q(w_1, y_1); {z_1/a_2})
(s(z_1, w_1), q(w_1, y_1); {z_1/a_3})
σ_5 = {x_2/a_2, w_1/y_2}
σ_6 = {x_2/a_2, w_1/y_2}
σ_7 = {w_1/y_2, y_2/b_3}
σ_8 = {x_2/a_3, w_1/y_2}
(r(x_2, y_2), q(y_2, y_1); {x_2/a_2})
(p(x_2, z_2), s(z_2, w_2), q(w_2, y_2), q(y_2, y_1); {x_2/a_2})
(q(y_2, y_1); {y_2/b_3})
(p(x_2, z_2), s(z_2, w_2), q(w_2, y_2), q(y_2, y_1); {x_2/a_3})
σ_9 = {z_2/a_3}
σ_10 = {z_2/a_1}
σ_11 = {y_1/b_2}
Fail
(s(z_2, w_2), q(w_2, y_2), q(y_2, y_1); {z_2/a_3})
(s(z_2, w_2), q(w_2, y_2), q(y_2, y_1); {z_2/a_1})
(□; { })
Fail
σ_12 = {x_3/a_3, w_2/y_3}
σ_13 = {x_3/a_3, w_2/y_3}
(r(x_3, y_3), q(y_3, y_2), q(y_2, y_1); {x_3/a_3})
(p(x_3, z_3) ...; {x_3/a_3})
σ_14 = {y_3/b_3}
Fail
(q(y_3, y_2), q(y_2, y_1); {y_3/b_3})
σ_15 = {y_2/b_2}
(q(y_2, y_1); {y_2/b_2})
σ_16 = {y_1/b_1}
(□; { })
p: p(a_1, y) / p(a_1, a_2), p(a_1, a_3)
p(a_2, y) / p(a_2, a_3), p(a_2, a_1)
q: q(b_3, y) / q(b_3, b_2)
q(b_2, y) / q(b_2, b_1)
r: r(a_1, y) / {}
r(a_2, y) / {}
r(a_3, y) / r(a_3, b_3)
s: s(a_1, y) / s(a_1, b_1), s(a_1, b_2)
s(a_2, y) / s(a_2, b_2)
s(a_3, y) / s(a_3, b_3)

Fig. 1. Search tree for Example 1.

The RILNs are the only nodes which cannot be expanded further in the expansion process. However, cutting off such a path in the graph may affect the completeness, because some solutions relying on this node cannot be evaluated. Therefore, a mechanism is needed to evaluate the remaining solutions in some way.

Definition 2.3. A repeated complete lookup node (RCLN) is a node which is subsumed by a previous node which has appeared earlier but not on the same path as the RCLN. For example, node $(s(z_1, w_1), q(w_1, y_1); \{z_1/a_3\})$ of the graph shown in fig. 1 is an RCLN because it is subsumed by $(s(z_2, w_2), q(w_2, y_2), q(y_2, y_1); \{z_2/a_3\})$ which has appeared earlier but not on the same path.

From the above definitions, we see that there are two kinds of subsumption checks which must be handled differently. When an RILN is encountered, the expansion should be suspended and the corresponding cycle should be recorded explicitly, while when an RCLN is encountered, it should be expanded immediately using the solutions already found. In addition, as we will see later, the ways in which RILNs and RCLNs are used to speed up the evaluation are different. However, distinguishing RCLNs from RILNs is not trivial and a more sophisticated technique is needed. To this end, we combine the technique for finding a topological order for a digraph with the technique for isolating the strongly connected components of a digraph (i.e. Tarjan's algorithm[15]) in such a way that the task can be done in linear time.

In what follows, we describe this method in detail.

Example 1. Axioms: (1) $s(x, y) :- r(x, y)$,

(2) $s(x, y) :- p(x, z), s(2, w), q(w, y)$.

Facts: $p(a_1, a_2), p(a_1, a_3), p(a_2, a_3), p(a_2, a_1)$,

$r(a_3, b_3)$,

$q(b_3, b_2), q(b_2, b_1)$.

Given the query ? - $s(a_1, y)$, the search tree depicted in fig. 1 will be generated by the OLDTNF resolution. The associated solution table is shown in the lower right part of figure 1.

Note that in fig. 1 the node $(s(z_1, w_1), q(w_1, y_1); \{z_1/a_3\})$ is an RCLN and the node $(s(z_2, w_2), q(w_2, y_2), q(y_2, y_1); \{z_2/a_1\}$ is an RILN. Because $(s(z_2, w_2), q(w_2, y_2), q(y_2, y_1); \{z_2/a_4\})$ is subsumed by node $(s(a_1, y); \{\})$ that has appeared earlier on the same path, we expect to extend a series of subgraphs similar to the first one from this node, which has already been traversed. Therefore, although the infiniteness can be avoided in an OLDTNF resolution by using tabulation, much redundant work may be done due to similar computations repeatedly performed. Thus, the traversal along a cyclic path has to be cut off if we do not want to do any useless work. However, cutting off a path may affect the completeness. We then have to record cycles explicitly and evaluate the corresponding solutions along the cycles in a subsequent phase. In contrast, each RCLN must be handled immediately to get some new solutions, which may be re-used in the subsequent traversal. A careful observation shows that the subgraph rooted at an RCLN is always similar to the one rooted at its subsuming node and this similarity can be employed to expedite the evaluation.

Below is a graph traversal algorithm which can isolate all cycles of a graph being traversed and at the same time can recognize all RCLNs of the graph in linear time. Combining this algorithm with techniques described in the next subsection, an optimal strategy for evaluating recursive queries can be obtained.

Obviously, for the above example, we do not need to check subsumptions over all nodes but only those whose head literal is with predicate name s. For this purpose, we introduce the notion of control graphs.

Definition 2.4. A control graph for an expansion process is a digraph where there is a node for each node (in a search tree) of the form ("$A_1, A_2, \cdots, A_n$", σ) with A_1 being a recursive predicate and an edge from node a to node b if and only if there is a path from a to b in the original search tree, which contains no other nodes with a head recursive predicate.

For example, the control graph of the expansion process shown in fig. 1 is as follows (figure 2):

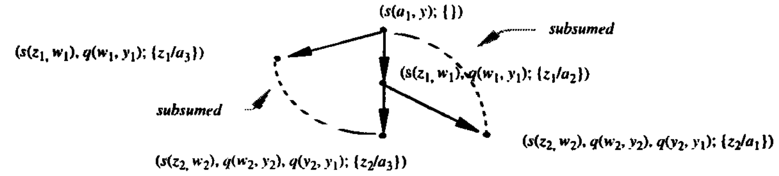

(s(a₁, y); {})
subsumed
(s(z₁, w₁), q(w₁, y₁); {z₁/a₃})
(s(z₁, w₁), q(w₁, y₁); {z₁/a₂})
subsumed
(s(z₂, w₂), q(w₂, y₂), q(y₂, y₁); {z₂/a₁})
(s(z₂, w₂), q(w₂, y₂), q(y₂, y₁); {z₂/a₃})

Fig. 2. Control graph.

The purpose of control graphs is to explain the control mechanism used in our method. In fact, it is sufficient to perform subsumption checks only on those nodes of a search tree, which appear also in its control graph (see next subsection). Therefore, we give the following algorithm over a control graph instead of a whole search tree so as to illustrate the key ideas clearly.

We associate each node v of a control graph with three integers dfsnumber(v), toplnumber(v) and lowlink(v). dfsnumber is used to number the nodes of a control graph in the order they are reached during the search. toplnumber is used to number the nodes with the property that all descendants of a node having toplnumber value m have a lower toplnumber value than m, i. e. a topological order numbering. It is used, here, to test whether a node is an RCLN. lowlink is used to number the nodes in such a way that if two nodes v and w are in the same strongly connected component, then lowlink(v) = lowlink(w). Therefore, it can be used to identify the "root" of a strongly connected component (a root is a node of a strongly connected component, which is first visited during the traversal). With the help of a stack structure, all strongly connected components can be feasibly found based on the calculation of lowlink values.

Essentially, the algorithm presented below is a modified version of Tarjan's algorithm[15]. The difference between them consists in the use of toplnumber in the modified algorithm, which facilitates the identification of a strongly connected component. (In the original algorithm, a stack structure must be searched to do this.) In addition, for our purposes, each RCLN and each RILN are marked.

```
procedure graph-algo(v) (* depth-first traversal of a graph rooted at v *)
  begin
    i := 0; j := 0; (* i and j are two global variables, used to calculate dfsnumber and
                       toplnumber, respectively. *)
    toplnumber (v) := 0;
    graph-search (v); (* go into the graph *)
  end
procedure graph-search (v)
  begin
    i := i + 1; dfsnumber (v) := i; lowlink (v) := i; (* initiate lowlink value; it may be
                                                         changed during the search *)
    put v on stack S; (* S is used to store strongly connected components if any *)
    generate all sons of v;
    for each son w of v do
      begin
          toplnumber (w) := 0; (* when a node is encountered at the first time, its
                                  toplnumber value is set to be 0. *)
      end
    for each son w of v do
      begin
        subsumption checking for w;
        if w is not subsumed by any node then
          begin
            call graph-search (w);          (* go deeper into the graph *)
            lowlink(v) := min(lowlink(v), dfsnumber(w)); (* the root of a subgraph
                                            will have the least lowlink value *)
          end
        else (* w is subsumed by some node *)
          {suppose that w is subsumed by u;
           if dfsnumber (u) < dfsnumber (v) then
            if toplnumber (u) > 0 then (* if u is typologically numbered, it cannot be an
                                        ancestor node of $v$. * )
                  mark $w$ to be an RCLN;
                else      ( * a cycle is encountered* )
                  {mark $w$ to be an RILN;
                  lowlink($v$) := min (lowlink($v$), dfsnumber ($u$)); } } ( * this operation will
                                        make all nodes of a strongly connected component have
                                        the same lowlink value as the root. * )
          end
        if (lowlink($v$) = dfsnumber ($v$)) then ( * $v$ is a root of some strongly connected com-
                                        ponent* )
          begin
            while $w$ on the stack $S$ satisfies dfsnumber ($w$) $\geqslant$ dfsnumber ($v$) do
                  {delete $w$ from the stack $S$ and put $w$ in current component (rooted at $v$);
                  toplnumber ($v$) := $j$; } ( * topological order numbering* )
            $j$ := $j$ + 1; ( * $j$ is used to calculate toplnumber* )
          end
      end
```

By a simple analysis, we know that this algorithm requires only linear time[15]. Fig. 3 shows a directed graph, its defnumber, lowlink and toplnumber values, and its strongly connected components.

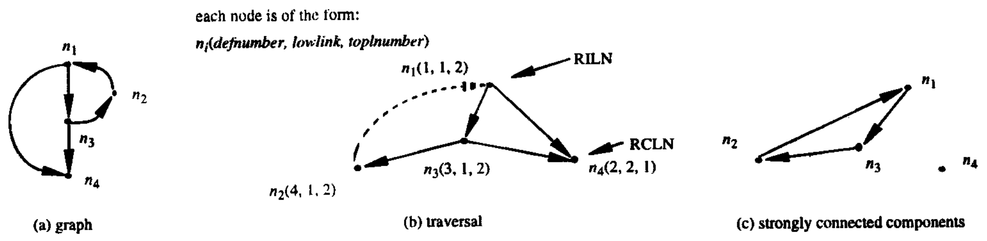


Fig. 3. Graph traversal.

2.2 Search space reduction and answer generation

Based on the mechanism for subsumption checks, we develop two methods for generating solutions by using the solution similitude property and the corresponding "path information": (i) solution generation for RCLNs; (ii) solution generation for RILNs.

(1) Solution generation for RCLNs

We clarify the first improvement by tracing the steps of the OLDTNF resolution for the following allowed program:

Example 2. Axioms: (1) $s(x, y)$: - $r(x, y)$,

(2) $s(x, y)$: - $p(x, z), s(z, w), t(y), \neg q(w, y)$.

Facts: $p(a_1, a_2), p(a_1, a_3), p(a_2, a_3)$,

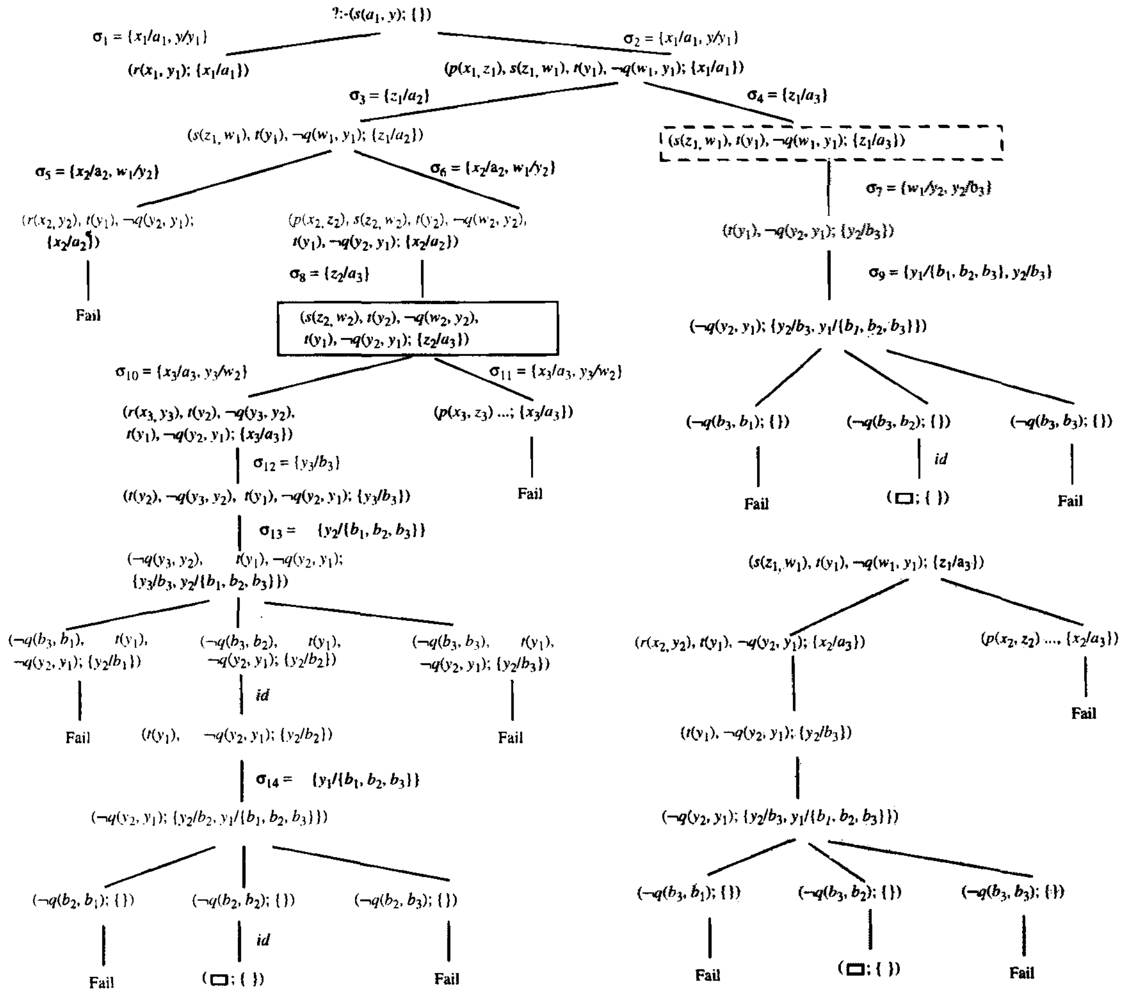
$r(a_3, b_3)$,

$t(b_3), t(b_2), t(b_1)$,

$q(b_3, b_3), q(b_3, b_1), q(b_2, b_3), q(b_2, b_2)$.

Given the query ? - $s(a_1, y)$, the search tree depicted in fig. 4 will be generated by the

OLDTNF resolution. The associated solution table is shown in figure 5.



?:-(s(a1, y); { })
σ1 = {x1/a1, y/y1}
σ2 = {x1/a1, y/y1}
(r(x1, y1); {x1/a1})
(p(x1, z1), s(z1, w1), t(y1), ¬q(w1, y1); {x1/a1})
σ3 = {z1/a2}
σ4 = {z1/a3}
(s(z1, w1), t(y1), ¬q(w1, y1); {z1/a2})
(s(z1, w1), t(y1), ¬q(w1, y1); {z1/a3})
σ5 = {x2/a2, w1/y2}
σ6 = {x2/a2, w1/y2}
σ7 = {w1/y2, y2/b3}
(r(x2, y2), t(y1), ¬q(y2, y1); {x2/a2})
(p(x2, z2), s(z2, w2), t(y2), ¬q(w2, y2), t(y1), ¬q(y2, y1); {x2/a2})
(t(y1), ¬q(y2, y1); {y2/b3})
σ8 = {z2/a3}
σ9 = {y1/{b1, b2, b3}, y2/b3}
Fail
(s(z2, w2), t(y2), ¬q(w2, y2), t(y1), ¬q(y2, y1); {z2/a3})
(¬q(y2, y1); {y2/b3, y1/{b1, b2, b3}})
σ10 = {x3/a3, y3/w2}
σ11 = {x3/a3, y3/w2}
(r(x3, y3), t(y2), ¬q(y3, y2), t(y1), ¬q(y2, y1); {x3/a3})
(p(x3, z3) ...; {x3/a3})
(¬q(b3, b1); { })
(¬q(b3, b2); { })
(¬q(b3, b3); { })
σ12 = {y3/b3}
id
Fail
(□; { })
Fail
(t(y2), ¬q(y3, y2), t(y1), ¬q(y2, y1); {y3/b3})
Fail
σ13 = {y2/{b1, b2, b3}}
(s(z1, w1), t(y1), ¬q(w1, y1); {z1/a3})
(¬q(y3, y2), t(y1), ¬q(y2, y1); {y3/b3, y2/{b1, b2, b3}})
(r(x2, y2), t(y1), ¬q(y2, y1); {x2/a3})
(p(x2, z2) ..., {x2/a3})
(¬q(b3, b1), t(y1), ¬q(y2, y1); {y2/b1})
(¬q(b3, b2), t(y1), ¬q(y2, y1); {y2/b2})
(¬q(b3, b3), t(y1), ¬q(y2, y1); {y2/b3})
id
Fail
Fail
(t(y1), ¬q(y2, y1); {y2/b2})
Fail
(t(y1), ¬q(y2, y1); {y2/b3})
σ14 = {y1/{b1, b2, b3}}
(¬q(y2, y1); {y2/b2, y1/{b1, b2, b3}})
(¬q(y2, y1); {y2/b3, y1/{b1, b2, b3}})
(¬q(b2, b1); { })
(¬q(b2, b2); { })
(¬q(b2, b3); { })
(¬q(b3, b1); { })
(¬q(b3, b2); { })
(¬q(b3, b3); { })
id
Fail
(□; { })
Fail
Fail
(□; { })
Fail

Fig. 4. Search tree for Example 2.

r:

$r(a_1, y)$	$r(a_2, y)$	$r(a_3, y)$
[ ]	[ ]	$r(a_3, b_3)$

s:

$s(a_1, y)$	$s(a_2, y)$	$s(a_3, y)$
$s(a_1, b_1)$	$s(a_2, b_2)$	$s(a_3, b_3)$
$s(a_1, b_2)$		

t:

$t(y)$
$t(b_1)$
$t(b_2)$
$t(b_3)$

p:

$p(a_1, y)$	$p(a_2, y)$
$p(a_1, a_2)$	$p(a_2, a_3)$
$p(a_1, a_3)$	

Fig. 5. Solution table for Example 2.

Observe that the node "$(s(z_1, w_1), t(y_1), \neg q(w_1, y_1); \{z_1/a_3\})$" (enclosed by a broken rectangle) in the graph shown in fig. 4 is an RCLN. It is because its head atom is the same as the head atom of the node "$(s(z_2, w_2), t(y_2), \neg q(w_2, y_2), t(y_1), \neg q(y_2, y_1); \{z_2/a_3\})$" (enclosed by a solid rectangle) which has appeared earlier but not on the same path. Thus, it can be resolved with the atoms in the corresponding solution table, i.e. the entry: $(s(a_3, y); s(a_3,$

b_3)) in the solution table shown in fig. 5. However, if we expand the search tree in a normal way, i.e. if we expand this node using the clauses of the program, a subtree like that shown in the lower left grey rectangle of fig. 4 will be traversed, which is similar to the subtree rooted at "($s(z_2, w_2), t(y_2), \neg q(w_2, y_2), t(y_1), \neg q(y_2, y_1); \{z_2/a_3\}$)" except that the latter is three levels higher than the former. Obviously, if a mechanism for recording the "path information" is provided, we can cut off the subtree rooted at "($s(z_1, w_1), t(y_1), \neg q(w_1, y_1); \{z_1/a_3\}$)" and generate the relevant solutions using its subsuming counterpart. Concretely speaking, if the keys of the relevant entries in the solution table shown in fig. 5 is linked in a way as shown in fig. 6 (a) (which corresponds to the control graph), we can generate the solutions: $s(a_1, b_2)$ and $s(a_3, b_3)$ directly (after the subtree rooted at "($s(z_2, w_2), t(y_2), \neg q(w_2, y_2), t(y_1), \neg q(y_2, y_1); \{z_2/a_3\}$)" is expanded) from the solutions already found and the associated path information instead of traversing the corresponding subtree (see fig. 6(b)). In this way, we can reduce the search space non-trivially.

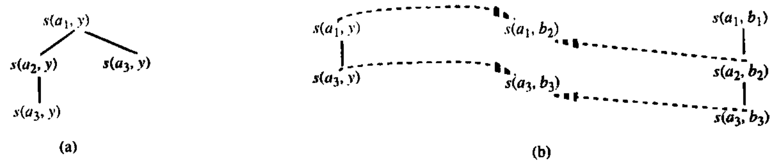


Fig. 6 Generation of solutions for RCLNs.

If the rules are reordered in the way as described at the beginning of this section, we can always feasibly generate the corresponding solutions for any subtree rooted at such a lookup node in terms of its subsuming counterpart and the associated path information. However, if the associated path is longer than the path of its subsuming counterpart, not all solutions for it can be generated and the remaining solutions must be evaluated by traversing the corresponding subtree. For instance, if we accidentally traverse the subtree rooted at "($s(z_1, w_1), t(y_1), \neg q(w_1, y_1); \{z_1/a_3\}$)" first, (producing the solutions: $s(a_1, b_2)$ and $s(a_3, b_3)$), we cannot generate all solutions for the subtree rooted at "($s(z_2, w_2), t(y_2), \neg q(w_2, y_2), t(y_1), \neg q(y_2, y_1); \{z_2/a_3\}$)". Thus, the remaining solutions must be evaluated by the standard method. Fig. 7 helps to clarify this claim.

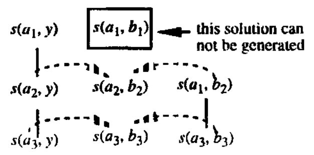


Fig. 7. Illustration for solution generation.

(2) Solution generation for RILNs

Now, we show another possibility of optimization for the above class of programs, when the input relations contain cyclic data. Consider the following example.

Example 3. Axioms: (1) $s(x, y) :- r(x, y)$,

(2) $s(x, y) :- p(x, z), s(z, w), q(w, y)$.

Facts: $p(c, d), p(c, b), p(b, c), p(b, f), p(f, c)$,

$r(d, e)$,

$q(e, a), q(a, i), q(i, o), q(o, g)$.

Given the query ?- $s(c, y)$, the search tree depicted in fig. 8 will be generated by the OLDTNF resolution. The associated solution table is shown in figure 9.

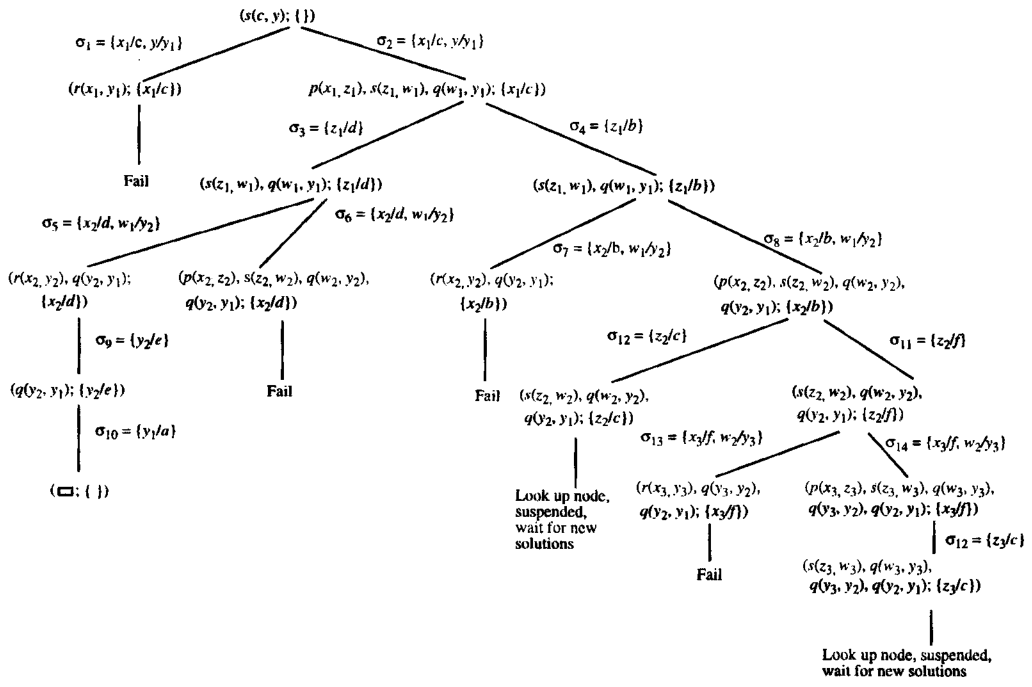

(s(c, y); { })
σ1 = {x1/c, y/y1}
σ2 = {x1/c, y/y1}
(r(x1, y1); {x1/c})
p(x1, z1), s(z1, w1), q(w1, y1); {x1/c})
σ3 = {z1/d}
σ4 = {z1/b}
Fail
(s(z1, w1), q(w1, y1); {z1/d})
(s(z1, w1), q(w1, y1); {z1/b})
σ5 = {x2/d, w1/y2}
σ6 = {x2/d, w1/y2}
σ7 = {x2/b, w1/y2}
σ8 = {x2/b, w1/y2}
(r(x2, y2), q(y2, y1); {x2/d})
(p(x2, z2), s(z2, w2), q(w2, y2), q(y2, y1); {x2/d})
(r(x2, y2), q(y2, y1); {x2/b})
(p(x2, z2), s(z2, w2), q(w2, y2), q(y2, y1); {x2/b})
σ9 = {y2/e}
σ12 = {z2/c}
σ11 = {z2/f}
(q(y2, y1); {y2/e})
Fail
Fail
(s(z2, w2), q(w2, y2), q(y2, y1); {z2/c})
(s(z2, w2), q(w2, y2), q(y2, y1); {z2/f})
σ10 = {y1/a}
σ13 = {x3/f, w2/y3}
σ14 = {x3/f, w2/y3}
(□; { })
Look up node, suspended, wait for new solutions
(r(x3, y3), q(y3, y2), q(y2, y1); {x3/f})
(p(x3, z3), s(z3, w3), q(w3, y3), q(y3, y2), q(y2, y1); {x3/f})
σ12 = {z3/c}
Fail
(s(z3, w3), q(w3, y3), q(y3, y2), q(y2, y1); {z3/c})
Look up node, suspended, wait for new solutions

Fig. 8. Search tree for Example 3.

r:	$r(c,y)$	$r(d,y)$		q:	$q(e,y)$	$q(a,y)$	$q(i,y)$	$q(o,y)$
	$[\,]$	$r(d,e)$			$q(e,a)$	$q(a,i)$	$q(i,o)$	$q(o,g)$
p:	$p(c,y)$	$p(b,y)$	$p(f,y)$	s:	$s(c,y)$	$s(b,y)$	$s(f,y)$	
	$p(c,d)$	$p(b,c)$	$p(f,c)$		$s(c,a)$	$s(b,i)$	$s(f,i)$	
	$p(c,b)$	$p(b,f)$			$s(c,o)$	$s(b,g)$	$s(f,g)$	
						$s(b,o)$		

Fig. 9. Solution table for Example 3.

Note that the node "$(s(z_2, w_2), q(w_2, y_2), q(y_2, y_1); \{z_2/c\})$" and the node "$(s(z_3, w_3), q(w_3, y_3), q(y_3, y_2), q(y_2, y_1); \{z_3/c\})$" are both RILNs. It is because both their head atoms are the same as the head atom of the node "$(s(c, y); \{\})$". Especially, they appear on the same path as "$(s(c, y); \{\})$", respectively. Therefore, the same goal will be derived infinitely many times if no other control mechanism is provided. During the execution of the OLDTNF resolution, such lookup nodes are checked and are resolved only with the solutions stored in the solution tables. If no solutions are available, the computation of these lookup nodes is suspended until new resolvable solutions are registered in the solution table.

The node "$(s(z_2, w_2), q(w_2, y_2), q(y_2, y_1), \{z_2/c\})$" can be resolved after the solution "$s(c, a)$" is stored in the corresponding solution table, producing the following solutions:

$$s(c, a), s(b, i), s(c, o), s(b, g).$$

Similarly, by resolving the node "$(s(z_3, w_3), q(w_3, y_3), q(y_3, y_2), q(y_2, y_1); \{z_3/c\})$", we will find another group of solutions:

$$s(a, c),\ s(f, i),\ s(b, o),\ s(c, g),\ s(f, g).$$

As demonstrated in Example 2, the subtree rooted at "$(s(z_2, w_2), q(w_2, y_2), q(y_2, y_1);$

$\{z_2/c\})$" does not need to be expanded, and the corresponding solutions can be generated directly in terms of the solutions already found if the relevant path information is recorded explicitly. The following figure illustrates this feature.

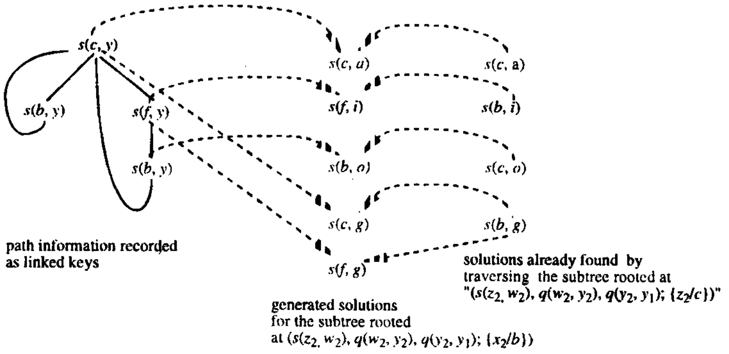

s(c, y)
s(b, y)
s(f, y)
s(b, y)
s(c, a)
s(c, a)
s(f, i)
s(b, i)
s(b, o)
s(c, o)
s(c, g)
s(b, g)
s(f, g)
path information recorded as linked keys
solutions already found by traversing the subtree rooted at "(s(z2, w2), q(w2, y2), q(y2, y1); {z2/c})"
generated solutions for the subtree rooted at (s(z2, w2), q(w2, y2), q(y2, y1); {x2/b})

Fig. 10. Generation of solutions for RILNs.

In this case, the control of the generation of solutions is a bit more complicated. We can imagine two circles (C_1 and C_2) with each corresponding to a cyclic path. We run respectively a-long C_1 and C_2 in the same direction and generate some solutions for C_2 from C_1 at each step. We do this continually until no new solutions for C_2 can be generated. Then we generate new solutions for C_1 from C_2 in the same way. This process is repeated until no more new solutions can be found.

3 Description of refined algorithm

From the above analysis, we know that in order to record the path information explicitly, a bit more complicated data structure must be provided. We do this by labeling each entry key of the solution table with an integer and at the same time adding to each entry key a field to store the pointers to those entry keys associated with its father nodes. In this way, we construct a graph which enables us to generate a lot of solutions without difficulty.

The refined algorithm can be described as follows.

As with OLDTNF resolution, the initial structure of the refined algorithm for $G\sigma$ is (Tr_0, Tb_0, As_0), where Tr_0 is a search tree consisting of only the root solution node labeled with (G, σ), Tb_0 is the solution table consisting of only one entry whose key is the head literal of the root node and whose solution list is an empty list [ ], and As_0 is an empty set of pointers.

Let T be an OLDTNF structure (Tr, Tb, As) in program P. An immediate extension of T by the refined algorithm is the result of the following operations, when node v of T is OLDTNF resolvable.

procedure expansion

(1) When v is a leaf solution node, let $C_1, C_2, \cdots, C_k (k \geqslant 1)$ be all the clauses with which the node v is OLDTNF resolvable, and (G_1, σ_1), (G_2, σ_2), $\cdots$, (G_k, σ_k) be the respective OLDTNF resolvents. Then add k child nodes of v labeled with $(G_1, \sigma_1), (G_2, \sigma_2), \cdots, (G_k, \sigma_k)$ to v. If the head atom of v is a recursive predicate (which is defined only by recursive clauses), construct the label for v and the pointers to its father nodes. In this way, the control graph can be constructed.

(2.1) When v is an RCLN (RCLNs can be identified using the algorithm introduced in subsec. 2.1), generate the solutions in terms of its subsuming counterpart and the associated path information. If the associated path is longer than the path of its subsuming counterpart, not all solutions for it can be generated and the remaining solutions must be evaluated by the standard method. We do this as follows. Let u be the last entry key on the path for which the solutions can be generated. Then, let $A\sigma\theta_1$, $A\sigma\theta_2, \cdots, A\sigma\theta_k (k \geqslant 1)$ be all (the variants of) the solutions in the associated solution list with which node u is OLDTNF resolvable, and (G_1, σ_1), (G_2, σ_2), $\cdots, (G_k, \sigma_k)$ be the respective OLDTNF resolvents. Then add k child nodes of u labeled with (G_1, σ_1), $(G_2, \sigma_2), \cdots, (G_k, \sigma_k)$ to u.

(2.2) When v is an RILN, mark the node as "suspended". No child node is generated.

(3) When v is a node corresponding to a root of some strongly connected component, call procedure "iteration" (see below).

(4) When v is a negative node ("$A_1, A_2, \cdots, A_n$", σ), let A be the head literal of the node and $A = \neg A_0$. We create a new initial OLDTNF structure T_0 of A_0, where its solution table is also newly created. We consider two cases. If there exists an extension of T_0 which contains a success leaf, then v has no child and it is a failure leaf. If there is a finitely failed OLDTNF structure which is an extension of T_0, then v has a unique child node v' of the form ("$A_2, \cdots, A_n$", σ). The edge from v to v' is labeled with identity substitution.

(5) In the cases of (1), (2.1) and (4), the edge from u to the node labeled with (G_i, σ_i) is labeled with θ_i, where θ_i is the substitution of the OLDTNF resolution. A new node is a lookup node when the head literal is a variant of some key in *Tb*. Otherwise, it is a solution node or a negative node. If a new node is an RCLN, add a pointer from the new lookup node to the head of the solution list of the corresponding key. If a new node is a solution node, add a new entry whose key is the head atom of the new node and whose solution list is an empty list.

(6) For each unit subrefutation of atom $A\sigma$ (if any) starting from a solution node and ending with some of the new nodes, add its solution $A\tau$ to the end of the solution list of $A\sigma$ in *Tb*, if $A\tau$ is not in the solution list.

In order to generate solutions for RILNs, the following procedure will be executed when a root of some strongly connected component is encountered during the expansion process.

procedure iteration

(1) Classify the cyclic paths such that all paths in a class has the same entry key. We find the cyclic paths in a search tree by recognizing the suspended node and tracing the pointers in the field added to the entry keys of solution table.

(2) For each class of cyclic paths, expand nodes only along one path.

(3) For each class of cyclic paths, generate solutions for all paths directly from the solutions already produced and the corresponding entry keys. This step is performed repeatedly until no new solutions can be generated (as described in subsection 2.1).

Example 4. Given the rules and facts as in Example 2. Let $q(a_1, x)$ be the query. The expansion process of the refined algorithm will produce the following solutions:

solutions evaluated by traversing the subtree rooted at "$((s(z_2, w_2), t(y_2), \neg q(w_2, y_2), t(y_1), \neg q(y_2, y_1); \{z_2/a_3\})$":

$$s(a_3, b_3), s(a_2, b_2), s(a_1, b_1),$$

answers generated for the subtree rooted at "$(s(z_1, w_1), t(y_1), \neg q(w_1, y_1); \{z_1/a_3\})$":

$$s(a_3, b_3), s(a_1, b_2).$$

The last two answers are generated directly from the solutions already found and the associated path information.

Example 5. Given the rules and facts as in Example 3. Let $q(c, x)$ be the query. The refined algorithm will produce the following solutions:

solutions evaluated in the expansion process: $s(d, e)$, $s(c, a)$;

solutions evaluated in the second step of the iteration process: $s(b, i)$, $s(c, o)$, $s(b, g)$;

solutions generated in the third step of the iteration process: $s(f, i)$, $s(b, o)$, $s(c, g)$, $s(f, g)$.

4 Correctness of the refined algorithm

In this section, we prove the correctness of the refined algorithm for programs which are reordered as described at the beginning of Section 2.

In order to prove the correctness of the refined algorithm, we have to specify that any solution produced in some subtree (of OLDTNF resolution) rooted at an RILN can be evaluated in the second step or generated in the third step of the iteration process of the refined algorithm. (For simplicity, we assume that RCLNs are handled in the same way that OLDTNF resolution does. So we can concentrate ourselves on the correctness of the treatment of RILNs. However, the correctness of refinement for RCLNs can be derived from the proofs of the following theorems.) For exposition, here we consider the case that there is only one set of the cyclic paths with a common node whenever the procedure "iteration" is called.

Theorem 1. *Let S be a solution produced in a subtree (of OLDTNF resolution) rooted at an RICN. Then S can be evaluated or generated in the iteration process of the refined algorithm.*

Proof. The proof of the theorem is by induction on the order, in which the solutions are produced in a subtree (of OLDTNF resolution) rooted at an RILN. The order is defined as follows. Let v be an RILN and $A_1, A_2, \cdots, A_k$ be all the solutions in the associated solution list (with which node v is OLDTNF resolvable) when v is first encountered. Then eval-order(A_1) = eval-order(A_2) = $\cdots$ = eval-order(A_k) = 0. These solutions may be utilized to produce new solutions (through expanding nodes), say $B_1, B_2, \cdots, B_m$. Then we have eval-order(B_1) = eval-order(B_2) = $\cdots$ = eval-order(B_k) = 1. In general, if a solution T is produced on another solution S, then eval-order(T) = eval-order(S) + 1.

Base Case: For each solution S with eval-order(S) being 1, S is produced along the first cyclic path in OLDTNF resolution and so can be evaluated in the second step of the iteration process of the refined algorithm.

Induction Step: Suppose that for some k, for all produced solutions with eval-order $\leqslant k$, they can be produced or generated in the iteration process of the refined algorithm and that eval-order(T_0) = $k+1$. We prove that T_0 can also be produced or generated in the iteration process of the refined algorithm. Let S_0 be the solution, on which T_0 is produced. Then eval-order(S_0) $\leqslant k$. By the induction hypothesis, S_0 can be produced or generated in the iteration process of the refined algorithm. Let S_1 be the solution, from which S_0 is generated and T be a set of solutions produced on the S_1. We prove that any solution produced on S_0 can be produced or generated from a solution in T. Since all literals occurring after a recursive predicate are only correlated with the recursive predicate and the bindings for their variables are completely determined by the bindings for certain variables of the recursive predicate, which do not appear as arguments of the literals occurring before the recursive predicate, the bindings for the unbound variables of the recursive predicate of the ith call are completely determined by the solutions to the recursive query of the $(i+1)$th call. That is, the bindings for the unbound variables of the recursive predicate in the $(j+1)$th step of some cyclic path is determined by the bindings for the unbound variables of the recursive predicate in the jth step of the cyclic path. Since S_0 is generated from S_1, they have the same bindings for the variables which are not bound to constants during the expansion. Therefore, a solution produced on S_0 will have the same bindings for the variables which are unbound during the expansion as some solution in T. Assume that $T_1 \in T$ has the same bindings for the unbound variables as T_0, then T_0 can be generated from T_1 in the iteration process of the refined algorithm. In the same way, we can find another solution, from which T_1 can be generated. In general, we can find a sequence $T_0, T_1, \cdots, T_m$ such that T_i is generated from T_{i+1} and eval-order$(T_m) = 0$ or eval-order$(T_m) \leqslant k$. By the induction hypothesis, T_m can be produced or generated in the iteration process of the refined algorithm. Therefore, T_0 can be produced or generated in the iteration process of the refined algorithm.

Theorem 2. *Let S be a solution generated in the iteration process of the refined algorithm. Then S can be produced by OLDTNF resolution.*

Proof. The proof of the theorem is also by induction on the order, in which some solutions are generated in the third step of the iteration process of the refined algorithm. The order is defined as follows. Let $A_1, A_2, \cdots, A_k$ be all the solutions evaluated in the second step of the iteration process of the refined algorithm. Then gen-order(A_1) = gen-order$(A_2) = \cdots =$ gen-order$(A_k) = 0$. If B is generated directly from some A_i (and the key of the corresponding solution list), then gen-order$(B) = 1$. In general, if a solution T is generated from another solution S (and the key of the corresponding solution list), then gen-order(T) = gen-order$(S) + 1$.

Base Case: For each solution S with gen-order(S) being 0, S is produced in the second step of the iteratoin process and so can be produced by OLDTNF resolution.

Induction Step: Suppose that for some k, for all generated solutions of gen-order$\leqslant k$, they can be produced by OLDTNF resolution and that gen-order$(S_0) = k+1$. We prove that S_0 can also be produced by OLDTNF resolution. Let T_0 be the solution from which S_0 is generated. Then gen-order$(T_0) \leqslant k$. By the induction hypothesis, T_0 can be produced by OLDTNF resolution. Let T_1 be the solution on which T_0 is produced and from which a set S is generated. Consider a solution $S_1 \in S$ on the same incomplete path as S_0. As demonstrated in the proof of Theorem 1, we know that S_0 can be produced on S_1 by OLDTNF resolution. In general, we can find a sequence S_0, S_1, $\cdots$, S_n such that S_i is produced on S_{i+1} and gen-order$(S_n) = 0$ or gen-order$(S_n) \leqslant k$. By the induction hypothesis, S_n can be produced by OLDTNF resolution. Therefore, S_0 can be produced by OLDTNF resolution.

5 Conclusion

In this paper, we present a top-down but set-oriented method, based on OLDTNF resolution, to handle recursive queries in allowed stratified databases. The key idea of the optimization is to distinguish between two classes of lookup nodes in an OLDTNF derivation and handling them in different ways. That is, we use the first class of lookup nodes to reduce the search space and develop an iteration process to generate most of the solutions for the second class of lookup nodes directly in terms of the path information and the solutions already found. In this way, we achieve high efficiency by minimizing the number of accesses to the external storage.

References

1 Lloyd, J. W., *Foundations of Logic Programming*, Berlin: Springer-Verlag, 1987.

2 Tamaki, H., Sato, T., OLD resolution with tabulation, in *Proc. 3rd Inter. Conf. Logic Programming*, Cambridge, London: MIT Press, 1986, 84—98.

3 Vieille, L., From QSQ to QoSaQ: Global optimization of recursive queries, in *Proc. 2nd Int. Conf. Expert Database System* (ed. Kerschberg, L.), Charleston, 1988, 743—748.

4 Seki, H., Itoh, H., A query evaluation method for stratified programs under the extended CWA, in *Proc. Inter. Conf. Logic Programming*, Cambridge, London: MIT Press, 1989, 195—211.

5 Balbin, G. S., Ramamobanarao, P. K., Menakshi, K., Efficient bottom-up computation of queries on stratified databases, *J. Logic Programming*, Amsterdam: Elsevier Science Inc., North-Holland, 1991, November, 295—344.

6 Chen, Y., Magic sets and stratified databases, *Int. J. Intelligent Systems*, New York: John Wiley & Sons Inc., 1997, 12 (3): 203.

7 Ullman, J., Van Gelder, A., Parallel complexity of logic programs, *Algorithmica*, Berlin: Springer-Verlag, 1988, 5—12.

8 Vardi, M. Y., The complexity of relational query language, *in Proc. 14th Annual ACM Symp. Theory of Computing*, San Francisco, Calif., May 5—7, 1982, New York: ACM, 1982, 137—146.

9 Aho, A. V., Sagiv, Y., Ullman, J. D., Efficient optimization of a class of relational expression, *ACM Trans. Datab. Syst*. 4.4 (Dec. 1979), New York, 1979, 435—454.

10 Gaifman, H., Mairson, H., Sagiv, Y. et al., Undecidable optimization problems for database logic programs, *J. ACM*, 1993, 40(3): 683.

11 Clark, K. L., Negation as failure, in *Logic and Databases* (eds. Gallaire, H., Minker, J.), New York: Plenum, 1978, 293—322.

12 Van Gelder, A., A message passing framework for logic query evaluation, in *Proc. 1986 CMSIGMOD Conf. On Management of Data*, New York: ACM, 1986, 155—165.

13 Cousot, P., Cousot, R., Abstract interpretation frameworks, *J. Logic Programming*, Amsterdam: Elsevier Science Inc., 1992, 2: 511.

14 Kanamori, T., Kawamura, T., Abstract interpretation based on OLDT resolution, *J. Logic Programming*, Amsterdam: Elsevier Science Inc., 1993, 15: 1.

15 Tarjan, R., Depth-first search and linear graph algorithms, *SIAM J. Compt*., 1972, 1(2): 146.