Controlling the Search in Bottom-Up Evaluation

Raghu Ramakrishnan
Divesh Srivastava
S. Sudarshan
Computer Sciences Department
University of Wisconsin–Madison
Madison, WI 53706, U.S.A.
{raghu,divesh,sudarshan}@cs.wisc.edu

Abstract

Bottom-up evaluation of queries on deductive databases has many advantages over an evaluation scheme such as Prolog. It is sound and complete with respect to the declarative semantics of least Herbrand models for positive Horn clause programs. In particular, it is able to avoid infinite loops by detecting repeated (possibly cyclic) subgoals. Further, in many database applications, it is more efficient than Prolog due to its set-orientedness. However, the completely set-oriented, breadth-first search strategy of bottom-up evaluation has certain disadvantages. For example, to evaluate several classes of programs with negation (or aggregation), it is necessary to order the inferences; in essence, we must evaluate all answers to a negative subgoal before making an inference that depends upon the negative subgoal. A completely breadth-first search strategy ([14]) would have to maintain a lot of redundant subgoal dependency information to achieve this.

We present a technique to order the use of generated subgoals, that is a hybrid between pure breadth-first and pure depth-first search. The technique, called Ordered_Search, is able to maintain subgoal dependency information efficiently, while being able to detect repeated subgoals, and avoid infinite loops. Also, the technique avoids repeated computation and is complete for DATALOG. We demonstrate the power of Ordered_Search through two applications. First, we show that it can be used to evaluate programs with left-to-right modularly stratified negation and aggregation more efficiently than with any previously known bottom-up technique. Second, we illustrate its use for optimizing single-answer queries for linear programs.

1 Introduction

Several studies ([11, 18, 3]) have shown similarities between different top-down evaluation methods and Magic Templates (or, Alexander Templates) based bottom-up evaluation methods for positive programs when all answers to a query are desired. In essence, the same subgoals and answers are generated by these methods when they use the same orderings of body literals in evaluating rules. However, there are important differences as well. In particular, the order in which subgoals and answers are generated and used in top-down evaluation strategies is different from the order in which they are generated and used in bottom-up evaluations. Top-down evaluations typically synchronize the generation of subgoals and answers to those subgoals, whereas bottom-up evaluations generate them asynchronously. This difference is not relevant for positive programs when all answers to a query are desired. However, when the program contains negation (or aggregation), the order in which inferences are performed becomes crucial to the correctness of the method, even when all answers to the query are desired. Again, when only a single answer to the query is desired, the order in which facts are generated and used becomes important, and the depth-first search strategy of a top-down evaluation scheme such as Prolog can perform much better than the breadth-first search strategy of bottom-up evaluation methods.

We describe a memoing technique called Ordered_Search that works on the transformed program obtained using Magic Templates rewriting, and is a hybrid between tuple-oriented top-down evaluation and set-oriented bottom-up evaluation. This technique generates subgoals and answers to subgoals asynchronously, as in bottom-up evaluation, while ordering the use of generated subgoals in a manner reminiscent of top-down evaluation. As a consequence, Ordered_Search is able to efficiently evaluate left-to-right modularly stratified programs [14] (see Sections 4.1 and 4.2), and restrict the search space in many cases when we want a single answer to the query (see Section 4.3).

1.1 Motivating Examples

Example 1.1 (Modular negation)
The class of programs with modular negation [14] naturally extends the class of programs with stratified and locally stratified negation while retaining a two-valued model. Consider the following left-to-right modularly stratified program-query pair $\langle P_{even}, Q_{even}\rangle$:

$r1 : even(X) :- succ(X, Y1), succ(Y1, Y), even(Y).$
$r2 : even(X) :- succ(X, Y), \neg even(Y).$
$r3 : even(0).$
$succ(1, 0).\ \ succ(2, 1).\ \ \ldots\ \ succ(n, n-1).$
Query: $?\text{-}\neg even(m).$

Ross [14] proposed a supplementary magic sets rewriting of $\langle P_{even}, Q_{even}\rangle$ in conjunction with a bottom-up method for evaluating the rewritten program. This method explicitly stores all the subgoal dependency information for negative subgoals. Ross' approach on this example would take $O(m^2)$ space and make $O(m^2)$ derivations since it would compute and store all the dependencies between subgoals transitively.

The technique presented in this paper, Ordered_Search, would compute and store only information about direct dependencies; hence, it would use $O(m)$ space and make $O(m)$ derivations in computing the query answer. (For more details, see Example 3.2.)

We describe other top-down and bottom-up techniques that can evaluate left-to-right modularly stratified programs in Section 5. As an example, the doubled program technique of Kemp et al. [7] would also use $O(m)$ space and make $O(m)$ derivations on this example. However, if rule $r1$ were removed from P_{even}, the doubled program approach would make $O(m^2)$ derivations, though it would still use only $O(m)$ space. Even on this modified program, Ordered_Search would compute the answer to the query using $O(m)$ space and making $O(m)$ derivations. □

Example 1.2 (Obtaining a single answer)
There are many cases where the user may want a single answer to a query. Consider, for example, the following program-query pair $\langle P_{path}, Q_{path} \rangle$.

$r1 : path(X, Y, [X, Y]) :- edge(X, Y).$
$r2 : path(X, Y, [X|P]) :- edge(X, Z), path(Z, Y, P).$
$edge(1, 2).$ $edge(1, 3).$ $edge(2, 1).$ $edge(2, 4).$ $edge(3, 4).$
Query: ?-$path(1, 4, X)$.

A top-down, tuple-oriented evaluation strategy, like Prolog, would set up a query on *path*, and solve the subgoals in a depth-first fashion. However, since there is a cycle in the *edge* relation, Prolog would not terminate on the given query.

One way of obtaining a single answer to the query is to evaluate the (magic transformed) program bottom-up until we get an answer to the query, and then terminate the evaluation. With this approach, subgoals are solved in parallel as they are generated.

The technique presented in this paper, Ordered_Search, solves subgoals in a depth-first fashion for this program, but since it performs memoing, it does not repeat computation and terminates on this program. In general, it provides an alternative evaluation strategy to the breadth-first strategy of bottom-up evaluation. For many programs (the above program with the given data is one such) a depth-first search for one answer is much more efficient than a breadth-first search for one answer. (For more details, see Example 3.1.) □

The rest of this paper is organized as follows. Preliminaries are covered in Section 2. The data structures and algorithms needed to evaluate a program using Ordered_Search are described in Section 3. We present results about the soundness, completeness, and efficiency of our procedure in Section 4. In Section 4.3, we characterize the order in which generated subgoals are selected to be used in terms of a depth-first traversal of the "subgoal-dependency" graph of the original program. Related work is described in Section 5, and directions for future research are indicated in Section 6.

2 Preliminaries

We assume familiarity with logic programming terminology (see [10]) and the issues involved in the bottom-up evaluation of logic programs. In particular, we assume the reader is familiar with Magic Templates rewriting ([11]), and with semi-naive bottom-up evaluation ([1]). For the purposes of this paper, a *program* is a set of *normal rules*. The techniques described in this paper are applicable to programs with uninterpreted function symbols, though for simplicity we restrict the programs to compute only ground facts.

We use the notion of a *subgoal-dependency graph* to characterize some of the results in this paper. Intuitively, the subgoal-dependency graph of a program-query pair is an AND/OR directed graph that characterizes the dependencies between subgoals set up in a top-down evaluation of the original program. Given a subgoal on the head of a rule, there are directed arcs in the subgoal-dependency graph to each subgoal set up during the evaluation of the body of that rule. We formalize this using SLP-trees and negation trees (see [14]) in the full version of the paper.

We assume the reader is familiar with the definition of (left-to-right) modularly stratified programs and the meaning of such programs (see [14]). Intuitively, a program is modularly stratified iff its mutually recursive components are locally stratified once all instantiated rules with a false subgoal that is defined in a "lower" component are removed. In the subgoal-dependency graph for left-to-right modularly stratified programs there is no cyclic dependency involving a negated subgoal. Ross' [14] technique as well as our technique makes essential use of this property in evaluating programs with left-to-right modularly stratified negation.

2.1 Modified Magic Templates Rewriting

Intuitively, the Magic Templates rewriting of a program defines a new predicate m_p (the magic predicate) for each predicate p in the original program P. The predicate m_p contains subgoals on p that need to be solved. Additional rules (derived from rules in P) that generate these subgoals are introduced in the rewritten program. Also, original program rules defining p are guarded by an m_p literal that ensures that only p facts matching the desired m_p subgoals are generated. The supplementary variant of Magic Templates avoids some recomputation by identifying common subexpressions, but at the cost of storing additional relations.

For the purpose of this paper, we modify the Magic Templates rewriting as follows: (1) For each (magic) predicate m_p in the Magic Templates transformed program P^{mg}, we create a new predicate $done\_m\_p$, which contains those subgoals on p all of whose answers have been computed. (2) For each rule R in P^{mg}, and for each negated literal, say $\neg q_i(\overline{t_i})$ in the body of R, we add the literal $done\_m\_q_i(\overline{t_i})$ to the body of R just before the occurrence of $\neg q_i(\overline{t_i})$.

Intuitively, the literal $done\_m\_q_i(\overline{t_i})$ will be satisfied only when the complete set of q_i answers matching $\overline{t_i}$ have been computed. Hence, this literal acts as a *guard* on the use of the subsequent negated q_i literal. In a similar fashion, we can also define the modified Supplementary Magic Templates rewriting. In the rest of this paper, we use SMT(P, Q) to refer to the program obtained by this modified Supplementary Magic Templates rewriting of program-query pair $\langle P, Q \rangle$, using left-to-right sips.

Further, when we talk about the dependencies between magic (or supplementary) facts in the rewritten program, we refer to the dependencies between subgoals in the original program, before the (Supplementary) Magic Templates rewriting has been performed.

3 Ordered Search

We now describe our evaluation technique, which we call Ordered_Search, that works on the transformed program obtained using Magic Templates or Supplementary Magic Templates rewriting. This technique generates subgoals and answers to subgoals asynchronously, as in bottom-up evaluation, but orders the use of generated subgoals in a manner reminiscent of top-down evaluation, and is in a sense a hybrid between pure (tuple-oriented) top-down evaluation and pure (set-oriented) bottom-up evaluation. We informally describe how Ordered_Search works on a transformed program-query pair $\langle P^{mg}, Q^{mg} \rangle$ and provide a detailed algorithmic description in the full version of the paper.

3.1 An Overview

The central data structure used by Ordered_Search, the *Context*, is used to preserve "dependency information" between subgoals. Ordered_Search can be understood as modifying semi-naive bottom-up evaluation as follows:

1. Newly generated magic and supplementary facts (if any) are inserted in the *Context* instead of being directly inserted in the differential relations. Consequently, these facts are hidden from the evaluation. (Other newly generated facts are inserted in the differential relations, and made available to the evaluation, as usual.)

2. Magic and supplementary facts from *Context* are *selectively* inserted into the differential relations (i.e., made available for further use by the evaluation) when no new facts can be derived using the current set of facts available to the evaluation, i.e., a fixpoint has been reached. (When a fact in *Context* is made available to the evaluation, it is said to be "marked" on the *Context*.)

3.2 Data Structures: *Context*

The *Context* is a sequence of *ContextNodes*. Each *ContextNode* has an associated set of magic facts and supplementary facts, and each magic or supplementary fact is associated with a unique *ContextNode*. A *ContextNode* is said to be "marked" if any magic or supplementary fact associated with the *ContextNode* is marked. The sequence of marked *ContextNodes* is a subsequence of the sequence of *ContextNodes*.

In the rest of this paper, when we use adjectives like "earlier", "later", etc. to refer to *ContextNodes* in *Context*, we mean their position in the sequence and not the time (which might be different) at which these nodes were inserted in the sequence.

We now intuitively describe the various operations performed on *Context*: (1) When a new magic or supplementary fact is inserted in *Context*, it is associated with a new *ContextNode*. Facts on *Context* are stored in an ordered fashion, such that if magic fact Q_1 generates (i.e., depends on) the magic fact Q_2, then Q_2 is stored after or along with Q_1 in the *Context*. (2) On detecting a cyclic dependency between subgoals on the *Context*, the associated *ContextNodes* are collapsed into one *ContextNode*, and all the facts associated with these *ContextNodes* are now kept together. Thus, unlike the stack of subgoals in Prolog evaluation, cyclic dependencies are handled gracefully. (3) When all the answers to a subgoal have been computed, the subgoal is removed from the *Context*.

3.3 Algorithms

We give an intuitive description of the Ordered_Search technique and in the process make several claims informally. These are formally stated and proved in the full version of the paper.

3.3.1 Inserting Facts into *Context*

Newly generated magic and supplementary facts (obtained by applying the semi-naive rules of the Magic transformed program) are inserted in the *Context* before they are selectively made available to the evaluation. When applying these rules, Ordered_Search records which magic or supplementary fact was used to make each derivation. (From the form of rules in the (Supplementary) Magic Templates transformation, there is exactly one such fact.) Let Q_1 be a newly computed magic/supplementary fact derived from magic/supplementary fact Q_2.

- If Q_1 is a magic fact $m\_p(\overline{t_1})$ that has been completely evaluated, it will be present in the $done\_m\_p$ relation.

  In this case, Ordered_Search does not insert Q_1 in *Context*.

- Else, since magic/supplementary facts that have been made available for use but have not been completely evaluated are marked in the

Context (see Section 3.3.2), we know that Q_2 occurs as a marked fact in a marked *ContextNode*.

The fact Q_1 is now inserted in a new unmarked *ContextNode* immediately before the next marked *ContextNode* following the marked *ContextNode* associated with Q_2 in the sequence of *ContextNodes*. (If there is no such marked *ContextNode*, Q_1 is inserted as the last *ContextNode* in the *Context*.) Thus, Q_1 is inserted after Q_2.

Since Q_2 depends on Q_1, "answers" to Q_1 could be used in computing "answers" to Q_2. Insertion, as above, is used to maintain dependency information between subgoals within the *Context* as a linear sequence. The order in which facts from *Context* are made available to the evaluation (see Section 3.3.2) will ensure that Q_1 is made available to the evaluation before Q_2 is said to be completely evaluated.

Duplicate elimination is now performed in the *Context* to ensure that there is at most one copy of Q_1 in *Context*. If there is more than one unmarked copy of Q_1 in *Context* at this stage, only the "last" copy of Q_1 is retained. If there is a marked copy of Q_1 in *Context*, i.e., if Q_1 has already been made available to the evaluation, there are two possibilities:

- If the marked copy of Q_1 occurs after the unmarked copy, only the marked copy of Q_1 is retained in *Context*.

- If the unmarked copy of Q_1 occurs after the marked copy, Q_1 depends on itself. We have thus detected a cyclic dependency between the set of all marked facts in *Context* in between the two occurrences of Q_1. Ordered_Search recognizes this and collapses this set of marked facts into the node of the marked copy of Q_1 in *Context*.

Collapsing marked facts into a single node when a cyclic dependency is detected is essential to the correctness of the technique in the presence of cycles in the subgoal-dependency graph of the original program. (Note that in left-to-right modularly stratified programs there can be positive cyclic dependencies, but no negative ones.) Since all these facts (cyclically) depend on each other, we cannot guarantee that any of these facts is completely evaluated until we know that all of them have been completely evaluated.

3.3.2 Making Facts Selectively Available

Facts from *Context* are made available to the evaluation only when no new facts can be computed using the set of available facts. If the last *ContextNode* contains at least one unmarked (magic or supplementary) fact, Ordered_Search chooses one such unmarked fact, marks it and makes it available to the evaluation by inserting it in the corresponding differential relation. (Note that this fact still remains in the *Context*.)

If all facts in the last *ContextNode* are marked, all the facts in the last *ContextNode* can be considered to be completely evaluated. Intuitively,

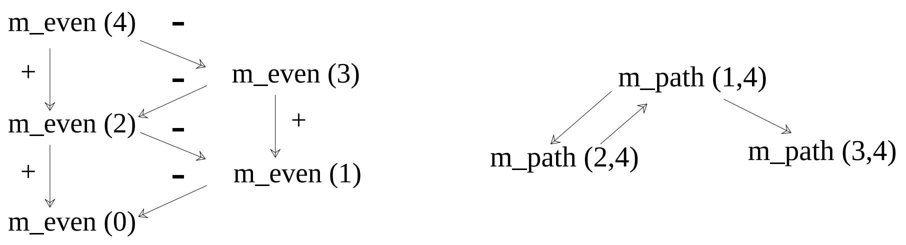

Figure 1: Subgoal Dependency Graphs for Motivating Examples

the reason for this is that a set of facts on $Context$ (that have been made available to the evaluation) can be considered to be *completely evaluated* if:

1. no new facts can be generated using the currently available set of facts (i.e., the iterative application has reached a fixpoint), and

2. every magic or supplementary fact generated from these facts has been completely evaluated.

All these facts are removed from $Context$ and all magic facts among these are inserted in the corresponding *done_m_p* relations. The last $ContextNode$ is now removed from $Context$. Thus, when a magic fact $m\_p(\overline{t_1})$ on $Context$ has been completely evaluated, it is moved to *done_m_p*.

3.4 Motivating Examples Revisited

We briefly describe how Ordered_Search can be used to evaluate the examples presented in Section 1.1.

Example 3.1 (Obtaining a single answer)
Consider the program-query pair $\langle P_{path}, Q_{path}\rangle$ from Example 1.2, where the user wants a single answer to the query. For this program-query pair, the subgoal-dependency graph is shown in Figure 1. Note that the subgoal-dependency graph has a cycle; consequently, Prolog would not terminate on this example program-query pair.

The Magic Templates transformed program $\langle P^{mg}_{path}, Q^{mg}_{path}\rangle$ is straightforward and we do not describe it further. We describe the evaluation of $\langle P^{mg}_{path}, Q^{mg}_{path}\rangle$ using Ordered_Search briefly in Table 1. Facts in $Context$ marked with an * indicate facts made available to the evaluation, and facts in $Context$ within { } indicate facts associated with a single $ContextNode$. Note that an answer is produced in iteration 3, as in the semi-naive bottom-up evaluation of $\langle P^{mg}_{path}, Q^{mg}_{path}\rangle$. However, the evaluation using Ordered_Search has computed fewer facts than would be computed by pure bottom-up evaluation. Also note that a cycle was detected since $m\_path(1,4)$ was derived from $m\_path(2,4)$, and this magic fact occurs with an * earlier in $Context$. As a result, in iteration 2, several nodes in $Context$ have been collapsed together. □

Iter	Facts in	Ordered_Search
0	$path$	$\{\}$
	$Context$	$m\_path(1,4)$
1	$path$	$\{\}$
	$Context$	$m\_path(1,4)^*, m\_path(3,4), m\_path(2,4)$
2	$path$	$\{path(2,4,[2,4])\}$
	$Context$	$\{m\_path(1,4)^*, m\_path(2,4)^*\}, m\_path(3,4)$
3	$path$	$\{path(2,4,[2,4]), path(1,4,[1,2,4])\}$
	$Context$	$\{m\_path(1,4)^*, m\_path(2,4)^*\} m\_path(3,4)$

Table 1: Ordered_Search evaluation of $\langle P^{mg}_{path}, Q^{mg}_{path}\rangle$

Example 3.2 (Modular negation)
Consider the left-to-right modularly stratified program P_{even} from Example 1.1, and the query $?\neg even(4)$. For this program-query pair, the subgoal-dependency graph is shown in Figure 1.

We omit the details of the Supplementary Magic Templates transformed program $\langle P^{mg}_{even}, Q^{mg}_{even}\rangle$. The evaluation of the supplementary magic program using Ordered_Search computes and stores only information about direct dependencies as a linear ordering of the magic and supplementary facts on $Context$; hence, the evaluation uses linear space and makes a linear number of derivations.

The technique described in [14] computes and stores the transitive dependencies in addition to the direct dependencies on this example; consequently, it would use quadratic space and make a quadratic number of derivations (of facts and dependencies). We omit details because of space limitations and describe this example in detail in the full version of the paper. $\Box$

4 Results about Ordered Search

All results in this section are applicable to programs with function symbols, except where stated otherwise. We also assume for simplicity that only ground facts are generated.

4.1 Results on Soundness, Completeness and Non-repetition

The key "lemma" to establish that Ordered_Search computes the well-founded model of a left-to-right modularly stratified program states that magic facts are moved from $Context$ to the corresponding $done\_m\_p$ relations only when these facts have been completely evaluated. The soundness result below then follows from the exhaustive nature of the evaluation and the correctness of the Supplementary Magic Templates rewriting with the $done\_m\_p$ literals as guards for negative body literals (referred to as SMT rewriting).

Theorem 4.1 *Suppose $\langle P, Q\rangle$ is a left-to-right modularly stratified program-query pair. An evaluation of* Ordered_Search(SMT(P, Q)) *is* sound *wrt the well-founded semantics of $\langle P, Q\rangle$.* $\Box$

Duplicate elimination of newly generated magic and supplementary facts in *Context* ensures that the evaluation does not repeat derivations.

Theorem 4.2 *Suppose $\langle P, Q\rangle$ is a left-to-right modularly stratified program-query pair. An evaluation of* Ordered_Search(SMT(P, Q)) *does not repeat derivations.* $\Box$

For programs with function symbols and negation, there is no effective procedure that can guarantee completeness in general. If there is an infinite sequence of subgoals, each depending on the next one in the sequence, and Ordered_Search chooses to explore such an infinite path, it may not compute an answer to the original query, even if one exists. Such paths cannot exist for DATALOG programs. Hence, we have:

Theorem 4.3 *Suppose $\langle P, Q\rangle$ is a left-to-right modularly stratified DATALOG program-query pair. An evaluation of* Ordered_Search(SMT(P, Q)) *terminates and is complete wrt the well-founded semantics of $\langle P, Q\rangle$.* $\Box$

In general, even if there are function symbols, Ordered_Search is complete wrt the well-founded semantics whenever it terminates.

4.2 Results about Space and Time Complexity

In maintaining an auxiliary data structure, the *Context*, Ordered_Search uses more space than ordinary semi-naive bottom-up evaluation (which only needs to maintain differential relations). However, there is no increase in asymptotic space complexity compared to other bottom-up evaluation strategies. Intuitively, this is because duplicate elimination on the *Context* guarantees that the same set of magic, supplementary and answer facts are computed by the various evaluation strategies, and the space used by *Context* is proportional to the space used by the magic and supplementary facts computed.

Note that Ross' technique may use asymptotically more space than Ordered_Search, since it stores transitive dependencies explicitly. For instance, in Example 1.1, Ross' algorithm uses $O(m^2)$ space, whereas Ordered_Search uses $O(m)$ space. Our technique for evaluating left-to-right modularly stratified programs is *strictly better* than the algorithm in [14], in terms of the asymptotic space complexity.

We now compare the asymptotic time complexity of Ordered_Search with other bottom-up evaluation strategies. For positive programs, it is easy to see that semi-naive bottom-up evaluation and Ordered_Search make the same set of inferences, although the order in which the inferences are performed may be different. Further, for left-to-right modularly stratified programs, it can be shown that Ordered_Search makes no more inferences than Ross' method. Note, however, that Ross' algorithm may make asymptotically more inferences than Ordered_Search since it computes transitive dependencies. For instance, in Example 1.1, Ross' algorithm to makes $O(m^2)$ inferences, whereas Ordered_Search makes $O(m)$ inferences.

In order to obtain the total time taken by the Ordered_Search evaluation in terms of the asymptotic cost of derivations, we need to obtain the cost of each derivation in the Ordered_Search evaluation. Unification of ground facts can be done in constant time using hash-consing for ground terms; indexing and insertion of ground facts in relations can also be done in constant time using hash based indexing (see [13]).

Hence, the cost of each derivation depends on the operations on $Context$, and several of these operations are operations on sets: finding the node corresponding to a fact, taking the union of facts associated with nodes on $Context$, and deleting entire sets of facts associated with a $ContextNode$. These operations can be efficiently implemented using the union-find technique [17], with an amortized cost of $O(\alpha(N))$ per operation, where N is the total number of these operations on $Context$, and $\alpha(N)$ is the inverse Ackermann function. Consequently, we have:

Theorem 4.4 *Let $\langle P, Q\rangle$ be a program-query pair.*

1. *If $\langle P, Q\rangle$ is positive, let the time taken (in terms of asymptotic derivation cost) to evaluate SMT(P, Q) in a bottom-up semi-naive evaluation be T. Then, an evaluation of* Ordered_Search(SMT(P, Q)) *takes time $O(T\alpha(T))$.*
2. *If $\langle P, Q\rangle$ is left-to-right modularly stratified, let the time to evaluate $\langle P, Q\rangle$ using Ross' algorithm be T. Then, an evaluation of* Ordered_Search(SMT(P, Q)) *takes time $O(T\alpha(T))$.* □

Since $\alpha(T)$ is very small even for very large values of T, Ordered_Search compares favorably in asymptotic (space and time) complexity both to semi-naive bottom-up evaluation for positive programs, and to Ross' evaluation of left-to-right modularly stratified programs. Note that Ross' method can be asymptotically worse than Ordered_Search, as Example 1.1 showed.

As a corollary to the above result, we can show that Ordered_Search takes no more time (asymptotically) than either semi-naive bottom-up evaluation or Ross' method, when the subgoal dependency graph is acyclic.

4.3 Results on Ordering Selection of Subgoals

Recall that bottom-up evaluation of a Magic Templates transformed program generates subgoals and answers to the subgoals as in a top-down evaluation, although the order in which these are generated in the bottom-up evaluation may be quite different from a top-down evaluation. By ordering the newly generated facts in *Context*, Ordered_Search makes facts selectively available to the evaluation in a manner considerably different from pure bottom-up evaluation. We now show that the order in which generated subgoals (magic facts) are selected to be used by Ordered_Search is related to a top-down evaluation.

Theorem 4.5 *Suppose $\langle P, Q\rangle$ is a left-to-right modularly stratified program-query pair. In an evaluation of* Ordered_Search(SMT(P, Q))*, the order in which magic facts are marked corresponds to a* depth-first traversal *(with marking) of the subgoal-dependency graph of $\langle P, Q\rangle$ starting from Q.* $\Box$

The order in which Prolog explores the subgoal-dependency graph also corresponds to a depth-first traversal, although Prolog does not "mark" nodes, and hence may repeat computation. After generating an answer for a subgoal generated from a rule literal, Prolog continues with the next rule body literal, before attempting to generate more answers for the first subgoal. Ordered_Search, on the other hand, generates all answers for the first subgoal before trying to solve subgoals generated from the next rule body literal. Consequently, Prolog may perform a lot less computation than Ordered_Search in obtaining a single answer to the query. For linear programs, however, delaying the availability of subgoals to the Ordered_Search evaluation does not delay the computation of the first answer to the query (because of the asynchronous way in which answers are generated).

We conjecture that Ordered_Search is most useful for computing single answers to a query for the class of linear programs that may have cyclic subgoals (and hence Prolog is not suitable).

5 Related Work

Ordered_Search compares favorably with other top-down and bottom-up methods for evaluating logic programs in the literature. In earlier sections, we have presented a detailed comparison with semi-naive bottom-up evaluation and with Ross' technique to evaluate left-to-right modularly stratified programs. We present a brief comparison with other techniques below.

Prolog: Ordered_Search is sound, complete for DATALOG and does not repeat derivations. Prolog is not complete even for DATALOG, and may repeat derivations. Also, Prolog does not evaluate the class of left-to-right modularly stratified programs correctly.[1] Although Ordered_Search does give a measure of control for single answer queries, the Prolog search strategy is still likely to be superior in this respect (except for the class of linear programs with a large number of repeated subgoals, or cyclic subgoals).

QSQR/QoSaQ and Extension Tables: Extension Tables [5] is similar to Prolog, except that it memos facts and subgoals and can detect loops.

[1]Of course, a meta-interpreter can be written using Prolog to evaluate such programs.

QSQR/QoSaQ [19, 20] is a top-down, memoing, set-oriented strategy that is closely related to bottom-up evaluation with Supplementary Magic rewriting. Like Prolog, these techniques cannot deal with left-to-right modularly stratified negation/aggregation. The tuple-oriented search strategy of the Extension Tables variant ET* is closer to Prolog, and may be more useful than Ordered_Search in some settings when single answers are desired, but it repeats computation.

Ross also describes how his approach can be used to adapt QSQR to deal with left-to-right modularly stratified negation. In this case as well, dependencies between subgoals are maintained transitively, and our previous comparisons also apply to this case.

Subquery Completion: A variant of QSQR, *subquery completion*, was described in [8] to deal with recursively defined aggregates. It uses the dependencies between subgoals maintained by QSQR to handle a class of acyclic programs with aggregation. However, this technique does not deal with programs that have cycles in the subgoal-dependency graph of a strongly connected component with aggregates (even if the cyclic dependency is only between positive subgoals). Ordered_Search allows positive cycles in the subgoal dependency graph, and deals with them by collapsing nodes in the *Context*, and declaring all the facts in a collapsed node to be completely evaluated once a fixpoint is reached. There is no analogue to this step in the technique of [8].

Techniques for computing the well-founded model: There are several query evaluation techniques in the literature that compute answers under the well-founded model. For example, WELL! [2] is based on global SLS-resolution; XOLDTNF [4] is an extension of OLDT resolution; GUUS [9] is based on the alternating fixpoint semantics; and the technique of Kemp et al. [7] is based on alternating fixpoint semantics and magic sets. The class of programs handled by these techniques is larger than that handled by Ordered_Search, but each of these techniques can repeat computation even for left-to-right modularly stratified programs. This can result in a loss of efficiency of evaluation.

There are other proposed techniques that control the order of inferences in a bottom-up evaluation in some way. Sloppy Delta Iteration [15] provides a way to "hide" facts until they are to be used. Techniques for hiding facts are used in [6, 16] to evaluate programs with aggregate operations efficiently. These results are only tangentially related to Ordered_Search since the (motivation as well as the nature of the) orderings considered are quite different.

6 Conclusions and Future Work

We presented a memoing technique, Ordered_Search, that is a hybrid between breadth-first and depth-first search. This technique can be used to efficiently evaluate left-to-right modularly stratified programs, and it is also useful in computing single answers to queries. Fully set-oriented computation causes problems for the evaluation of left-to-right modularly stratified programs, as illustrated by our comparisons with Ross [14]—it can result in an order of magnitude slow-down. Hence, it is important to provide some of the benefits of tuple-at-a-time computation with bottom-up evaluation, and Ordered_Search does just this.

Ordered_Search can also be used for programs that compute non-ground facts; details are presented in the full version of the paper. Also, while our claims about correctness of Ordered_Search have been made for the class of left-to-right modularly stratified programs, we conjecture that the method is correct whenever there is no cyclic negative dependency in the subgoal dependency graph. We believe that Ordered_Search is a versatile and very useful tool in the evaluation of queries on deductive databases. Ordered_Search has been implemented in the CORAL system [12], and performance numbers will be presented in the full version of the paper.

An important direction of future research is to explore the possibility of increasing the set-orientedness of Ordered_Search, thereby increasing efficiency of evaluation, while retaining its desirable properties for evaluating left-to-right modularly stratified programs. Another direction of research is to provide a finer grain of control in making subgoals available to the evaluation such that the technique can mimic Prolog more closely, providing further benefits for queries requiring a single answer.

Acknowledgements

We would like to thank Alexandre Lefebvre for valuable discussions regarding QSQR and EKS-V1, and pointing out some related work. We would also like to thank the referees for several valuable suggestions. This research was supported by a David and Lucile Packard Foundation Fellowship in Science and Engineering, a Presidential Young Investigator Award, with matching grants from Digital Equipment Corporation, Tandem and Xerox, and NSF grant IRI-9011563.

References

[1] I. Balbin and K. Ramamohanarao. A generalization of the differential approach to recursive query evaluation. *Journal of Logic Programming*, 4(3), September 1987.

[2] N. Bidoit and P. Legay. WELL! An evaluation procedure for all logic programs. In *Proceedings of the International Conference on Database Theory*, pages 335–348, Paris, France, December 1990.

[3] F. Bry. Query evaluation in recursive databases: Bottom-up and top-down reconciled. *IEEE Transactions on Knowledge and Data Engineering*, 5:289–312, 1990.

[4] W. Chen and D. S. Warren. A practical approach to computing the well founded semantics. In *Proceedings of the Joint International Conference and Symposium on Logic Programming*, 1992.

[5] S. W. Dietrich. Extension tables: Memo relations in logic programming. In *Proceedings of the Symposium on Logic Programming*, pages 264–272, 1987.

[6] S. Ganguly, S. Greco, and C. Zaniolo. Minimum and maximum predicates in logic programming. In *Proceedings of the ACM Symposium on Principles of Database Systems*, 1991.

[7] D. Kemp, D. Srivastava, and P. Stuckey. Magic sets and bottom-up evaluation of well-founded models. In *Proceedings of the International Logic Programming Symposium*, pages 337–351, San Diego, CA, U.S.A., Oct. 1991.

[8] A. Lefebvre. Towards an efficient evaluation of recursive aggregates in deductive databases. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, June 1992.

[9] N. Leone and P. Rullo. Safe computation of the well-founded semantics of Datalog queries. *Information Systems*, 17(1):17–31, 1992.

[10] J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 1984.

[11] R. Ramakrishnan. Magic Templates: A spellbinding approach to logic programs. In *Proceedings of the International Conference on Logic Programming*, pages 140–159, Seattle, Washington, August 1988.

[12] R. Ramakrishnan, D. Srivastava, and S. Sudarshan. CORAL: Control, Relations and Logic. In *Proceedings of the International Conference on Very Large Databases*, 1992.

[13] R. Ramakrishnan and S. Sudarshan. Top-Down vs. Bottom-Up Revisited. In *Proceedings of the International Logic Programming Symposium*, 1991.

[14] K. Ross. Modular Stratification and Magic Sets for Datalog programs with negation. (A shorter version appeared in the Proceedings of the ACM Symposium on the Principles of Database Systems, 1990), 1991.

[15] H. Schmidt, W. Kiessling, U. Güntzer, and R. Bayer. Compiling exploratory and goal-directed deduction into sloppy delta iteration. In *IEEE International Symposium on Logic Programming*, pages 234–243, 1987.

[16] S. Sudarshan and R. Ramakrishnan. Aggregation and relevance in deductive databases. In *Proceedings of the Seventeenth International Conference on Very Large Databases*, Sept. 1991.

[17] R. E. Tarjan. *Data Structures and Network Algorithms*. Society for Industrial and Applied Mathematics, 1983.

[18] J. D. Ullman. Bottom-up beats top-down for Datalog. In *Proceedings of the Eighth ACM Symposium on Principles of Database Systems*, pages 140–149, Philadelphia, Pennsylvania, March 1989.

[19] L. Vieille. Recursive axioms in deductive databases: The query-subquery approach. In *Proceedings of the First International Conference on Expert Database Systems*, pages 179–193, Charleston, South Carolina, 1986.

[20] L. Vieille. From QSQ towards QoSaQ: Global optimizations of recursive queries. In *Proc. 2nd International Conference on Expert Database Systems*, Apr. 1988.