PROCESSING RECURSION IN DATABASE SYSTEMS

by

Yannis E. Ioannidis

Memorandum No. UCB/ERL M86/69

2 September 1986

PROCESSING RECURSION IN DATABASE SYSTEMS

by

Yannis E. Ioannidis

Memorandum No. UCB/ERL M86/69

2 September 1986

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

PROCESSING RECURSION IN DATABASE SYSTEMS

by

Yannis E. Ioannidis

Memorandum No. UCB/ERL M86/69

2 September 1986

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Processing Recursion in Database Systems

Yannis E. Ioannidis

Processing Recursion in Database Systems

Ph.D. *Yannis E. Ioannidis* Computer Science Division
Department of EECS

Prof. Eugene Wong
Committee Chairman

ABSTRACT

Deductive databases are those in which new facts or rules may be derived from facts or rules that have been explicitly introduced. An important addition when augmenting a relational database system with deductive capabilities is the ability to define relations recursively.

Horn clauses have been the traditional framework for the study of recursion. We embed the relational algebra operators that are equivalent to Horn clauses into a *closed semiring*. In this setting, linear immediate recursion is formulated as a linear operator equation. Answering queries on relations defined by recursive Horn clauses (recursive queries) is equivalent to solving the corresponding linear equations. Given a linear recursive Horn clause, the solution to the equivalent equation is the transitive closure of the operator corresponding to the Horn clause. Likewise, linear mutual recursion is formulated as a linear system of operator equations. Equivalence of linear and bilinear recursive Horn clauses is also addressed, and sufficient conditions are given in terms of the corresponding operators.

Recursion raises new computational challenges to a database system, largely because a recursive query is equivalent to an infinite number of nonrecursive ones. *Uniformly bounded* recursive Horn clauses are those that are equivalent to a fixed finite number of nonrecursive ones. We give necessary and sufficient conditions for Horn clauses in a specific class to be uniformly bounded.

The power of the algebraic formulation of recursion lies in the ability to manipulate an explicit representation of the operator equivalent to the query answer. Taking advantage of that, we propose several new algorithms to answer recursive queries, whose main characteristic is issuing fewer but more expensive nonrecursive queries than the traditional ones. Analytical and experimental results show that the new algorithms perform best for small relations.

Since no query processing algorithm is universally optimal, optimization becomes necessary. We devise two optimization algorithms, one based on *simulated annealing* and another based on *dynamic programming*. Both explore a state space of the equivalent forms of the transitive closure of relational operators. Experiments with the former show that the cost of the various query processing algorithms varies significantly, thereby emphasizing the importance of optimization.

Acknowledgements

The unconditional love of my parents and my sister since the first years of my life has been the only source of my strength that made this dissertation possible. Their inspiration, encouragement, and emotional support through happy and sad times has helped me to overcome all the obstacles. I offer this work to them as a small token of my appreciation for what they have been to me. Dora Tsitsiliani lighted my life with her love and understanding. Her brief passage through life was enough to reveal those rare human qualities that make living worth while. I wish to express my deep gratitude to her for what she revealed to me and wish her memory to be eternal.

Having Professor Gene Wong as my scientific father has been one of the brightest aspects of my studies at Berkeley. With a remarkable balance between judiciously advising me and letting me be independent, he has been the perfect guide for my research. His inexhaustible source of ideas and his insight into a wide range of problems has been of invaluable help to me. I wish to express my deep appreciation to him for providing me with the research model to which I aspire.

I would also like to express my gratitude to Professor Michael Stonebraker for his valuable criticism and encouragement during the course of my research as well as Professor Larry Rowe for his generous help in several problems. Special thanks go to Professor Phil Bernstein for putting the effort to make the transition from Harvard to Berkeley a reality. I would also like to thank the members of my thesis committee Professors Richard Karp and Jack Silver for reading the manuscript of my dissertation and providing me with several suggestions for improvement.

My life at Berkeley would be completely different had I not come here with Timos Sellis. Sharing my first research steps with him, ever since we were in Greece, has been an enriching experience. His input is to be found in all parts of this dissertation and I want to thank him for that. Our friendship, however, goes far beyond the scientific realm. I am grateful for having him as my "twin brother" and I hope that our geographic separation will be temporary.

I would like to give thanks to all my friends that shared my life for the past three years in Berkeley, especially Marilena Sellis, Voula Pandazopoulou, Takis Konstantopoulos, Theologos Kelessoglou, Costas Papamichael, Irene Papamichael, and Pandelis Tsoucas. I also wish to express my deep appreciation to Hoai Cao for her valuable friendship.

My friends and colleagues in the INGRES group deserve special thanks for creating this wonderful working environment. Especially, I want to thank Margaret Butler, Oliver Gunther, Eric Hanson, Stephane Lafortune, Margie Murphy, Brad Rubinstein, Toni Guttman, and many others that were part of "the turtles" in the past three years. I also want to thank Lorna Shinkle for our limited yet fruitful collaboration.

Finally, I would like to acknowledge the financial support I have received from the National Science Foundation under grant ECS-8300463.

TABLE OF CONTENTS

LIST OF FIGURES

4. PROCESSING ALGORITHMS

CHAPTER 1

INTRODUCTION

1.1. Expert and Logic Programming Systems

Expert Systems are problem-solving systems that solve substantial problems conceded as being difficult and requiring expertise [Stef82]. They are called ***knowledge based*** because their performance depends critically on the use of facts and heuristics used by experts, and the construction of them is referred to as ***knowledge engineering*** [Feig77].

The generic architecture of an expert system has two interacting components: a ***knowledge base*** and an ***inference engine*** [Clif83, Haye85]. This is shown in Figure 1.1.

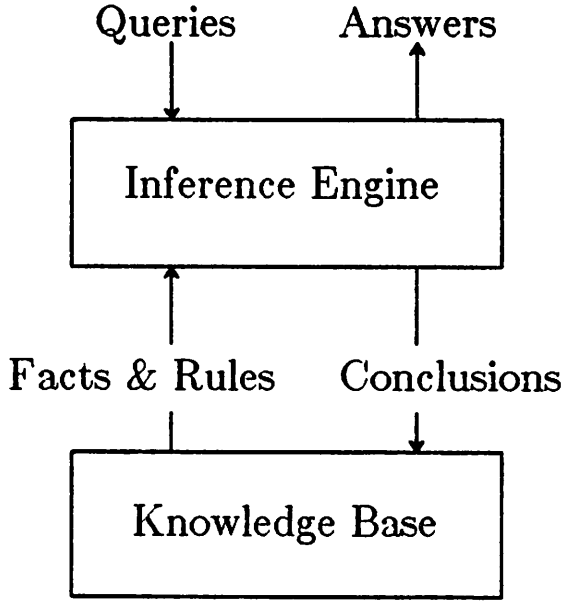

Figure 1.1. Generic architecture of an expert system.

The success of an expert system depends largely on the ability of the inference engine to exploit the knowledge base and perform inferences. Choosing the right representation and organization for the knowledge base affects the inference capabilities of the system significantly, so it is critical to the design of the system. The experience from the research of the past fifteen years on the subject has shown that the knowledge base is best represented declaratively, as opposed to procedurally. As a result of that, the most widely used representation for the knowledge base is a set of facts and a set of rules. Rules always express a conditional, with an antecedent and a consequent. Their interpretation is that if the the antecedent is satisfied, then the consequent is also.

Expert systems achieve good performance by specialization. Since an expert system is directed to a narrow domain of applications, the specific representation of the knowledge base and possibly the structure of the inference engine are specialized to serve the needs and peculiarities of this domain. This strength of an expert system is at the same time its weakness; it cannot be used for anything outside its application domain. *Logic Programming Systems* [Kowa83] take a more moderate approach. The generic architecture of expert systems and logic programming systems is the same (see Figure 1.1). The significant difference is that, in a logic programming system, both the structure of the inference engine and the representation of the knowledge base are application-independent. Inevitably, a logic programming system sacrifices some performance for wider applicability. As the name suggests, logic programming systems are usually based on first order logic [Ende72], and they employ logical inference rules (e.g. *modus ponens* [Ende72], *resolution* [Robi65]) in the inference engine

- Using a logic programming system as a front end to a database system. Even though at some abstract level logic programming systems and relational database systems are highly compatible, there are significant difficulties in coupling the two together. Certain aspects of the one system have no counterpart in the other (e.g. there is a significant semantic content in the order of clauses in Prolog, which is absent from database systems). This mismatch makes the task of integrating the two highly nontrivial. However, this approach has been tried out in many cases with noticeable success [Nico83, Jark84].

- Enhancing an existing logic programming system with database functionality. Even though work has been done to partially add database functionality to logic programming systems [Scio84], doing so to an absolute level requires writing a large portion of a database system, which accounts for a considerable amount of work.

- Enhancing an existing relational database system with inferential capabilities. This approach is the dual of the previous one and is taken by many researchers [Ioan84, Daya84]. Besides inferencing, there are few other services provided by logic programming systems that are absent from database systems, e.g. support for lists in Prolog [Park86]. We believe, however, that this is the approach that will result in the most effective deductive database systems.

While most of the analysis in this dissertation is applicable to all three system architectures, our primary conceptual framework is provided by the third approach.

and predicate calculus for the representation of the knowledge base.

Existing expert systems and logic programming systems usually deal with knowledge bases of small size. They function based on the assumption that the knowledge base is stored in virtual memory. In addition, they provide only limited services for concurrent access to distributed knowledge bases, recovery, protection, etc., if at all. This makes them inadequate for supporting many new knowledge-intensive applications, such as sophisticated office automation, computer aided design and manufacturing, decision support, etc. This lack of functionality on the part of expert systems and logic programming systems has motivated the development of *Expert Database Systems* and *Deductive Database Systems*. Expert database systems incorporate the functionality of both expert and database systems. Likewise, deductive database systems incorporate the functionality of both logic programming and database systems.

The focus of this dissertation is inference, independent of any particular application domain. Hence, the presentation is restricted to deductive database systems. A fairly complete picture of the research effort on expert database systems can be found in [Kers86].

1.2. Deductive Database Systems Architectures

Designing and building deductive database systems have taken many forms [Gall78, Gall81a, Dahl82, Gall84, Kowa84]. A significant role in this effort is played by the Prolog programming language [Cloc81] and systems based on it. There are three principal trends in integrating logic programming and database systems:

1.3. Deductive Databases and First Order Logic

We begin with some definitions in first order logic [Ende72], which will serve as a framework for the formulation of problems in deductive databases.

Definition 1.1: Consider a well-formed formula in a first order language. The formula is equivalent to a *Horn Clause* if it is of the form

$$\mathbf{R}_1(\underline{t}^{(1)}) \wedge \mathbf{R}_2(\underline{t}^{(2)}) \wedge \cdots \wedge \mathbf{R}_k(\underline{t}^{(k)}) \rightarrow \mathbf{R}_{k+1}(\underline{t}^{(k+1)}) \quad k \geq 0, \tag{1.1}$$

where $\mathbf{R}_i$ and $\underline{t}^{(i)}$, $1 \leq i \leq k+1$ are predicate symbols and vectors of terms in the language respectively [Ende72]. In addition, all the variables appearing in the formula are (implicitly) universally quantified.

The formulas to the left and right of $\rightarrow$ are called the *antecedent* or *qualification* and the *consequent* of the Horn clause respectively. Notice that $\mathbf{R}_i$ may be one of the *primitive* predicates symbols $=, \geq, >, \leq, <$. In any other case R_i is called a *database* predicate symbol.

Example 1.1: The following first order formula is a Horn clause

$$\mathbf{parent}(x,z) \wedge \mathbf{parent}(z,y) \rightarrow \mathbf{grandparent}(x,y),$$

whereas the next one is not

$$\mathbf{parent}(x,y) \rightarrow \mathbf{father}(\mathrm{x},\mathrm{y}) \vee \mathbf{mother}(\mathrm{x},\mathrm{y}). \qquad \square$$

Definition 1.2: A Horn clause is called *function-free* if it contains no function symbol. In other words, all the terms in (1.1) are either variables or constants.

Definition 1.3: A Horn clause is called *range-restricted* if all its variables appear under at least one database predicate symbol in the qualification.

Example 1.2: The first Horn clause below is range-restricted, whereas the last two are not.

$$\mathbf{parent}(x,z) \wedge \mathbf{parent}(z,y) \rightarrow \mathbf{grandparent}(x,y)$$

$$\mathbf{likes}(x,x) \rightarrow \mathbf{likes}(x,y)$$

$$x \geq 6 \rightarrow \mathbf{too\_many\_children}(x)$$ □

Definition 1.4: A Horn clause (1.1) is called *recursive* if $\mathbf{R}_{k+1} \in \{\mathbf{R}_1, \mathbf{R}_2, \ldots, \mathbf{R}_k\}$. In this case, $\mathbf{R}_{k+1}$ is called the *recursive predicate*, whereas any other predicate is called *nonrecursive*.

Definition 1.5: A recursive Horn clause is called *linear* if its recursive predicate appears only once in the antecedent.

Example 1.3: The two Horn clauses below are recursive, but only the first is linear.

$$\mathbf{ancestor}(x,z) \wedge \mathbf{father}(z,y) \rightarrow \mathbf{ancestor}(x,y)$$

$$\mathbf{brother}(x,z) \wedge \mathbf{brother}(z,y) \rightarrow \mathbf{brother}(x,y)$$ □

A deductive database is a relational database [Codd70] enhanced with a set of Horn clauses. Taking a model theoretic approach, all the predicate symbols that *do not* appear in the consequent of any Horn clause in the database are *interpreted* by the corresponding relations in the database [Nico78]. These constitute the *Extensional Database* (EDB). The predicates that *do* appear in the consequent of some Horn clause constitute the *Intentional Database* (IDB). For any given IDB there are many interpretations. If the IDB is defined by a set of Horn clauses, however, then it has a minimal interpretation [VanE76]. This is exactly the interpretation assumed in

a deductive database. It is deducing the interpretations of predicate symbols in the IDB that gave the name "deductive" to these systems. Most importantly, there exists an algorithm (based on resolution and unification [Robi65]) to answer any query presented to a Horn clause deductive database. Whenever it creates no confusion, instead of the term "predicate symbol", the term "relation" (its interpretation) is used.

Certain forms of Horn clauses do exist in ordinary relational databases as well [Ston76,Astr76]. In particular, *integrity constraints* and *view definitions* can be thought of as Horn clauses. The semantics of integrity constraints and views are different from the ones of Horn clauses though. Integrity constraints have a primitive relation in the consequent (i.e. one of $\{=,\neq,\geq,>,\leq,<\}$), which is interpreted by an algorithm within the system. In that sense, integrity constraints are not used for deductions; they simply have to be satisfied by the database contents at all times. Being interested in the deductive aspects of these systems we discuss integrity constraints no further.

View definitions have more restrictive semantics than Horn clauses. They do have a non-primitive predicate symbol in their consequent, so they are used to deduce information, but neither can there be two view definitions defining the same view, nor can a view have a stored part in the database. This implies also that there can be no nonempty views defined recursively. Finally, there is a certain semantic load on views concerning updates. Updates on a view are propagated to the underlying relations, whenever possible. This treatment of updates on views has no counterpart in Horn

clauses.

Despite the above, with respect to their deductive aspects only (no updates), views are similar to Horn clauses. They are usually processed using query modification [Ston75], which is a special form of resolution, potentially more efficient than the general algorithm [Dwor84]. In addition, views with multiple definitions and/or a stored part can be accommodated in a regular database system with only minimal enhancements. This is because all the queries on these extended types of views can still be transformed into a finite set of queries on the EDB.

What makes the real difference between view definitions and Horn clauses (or ordinary relational databases and deductive databases, for that matter) is that Horn clauses can be recursive. In general, queries on relations defined by a recursive Horn clause are equivalent to an infinite number of queries on the EDB. This creates new computational challenges to the deductive database system, distinctly different from those of relational database systems. We believe that identifying efficient ways to process queries on recursively defined relations is the key issue in building usable deductive database systems. This is exactly the focal point of this dissertation.

Example 1.4: Consider a deductive database whose EDB consists of a binary relation **father** with schema **father**(fath,son). In addition, the IDB consists of the following two Horn clauses, which define the binary relation **ancestor** with schema **ancestor**(anc,desc):

$$\mathbf{ancestor}(x,z) \wedge \mathbf{father}(z,y) \rightarrow \mathbf{ancestor}(x,y)$$

$$\mathbf{father}(x,y) \rightarrow \mathbf{ancestor}(x,y).$$

Let **ancestor**(x,*Ermis*)? be a query given to the system, asking for Ermis' ancestors. Resolution theorem proving [Robi65], or query modification [Ston75] gives the following infinite number of queries:

$$\textbf{father}(x,\textit{Ermis})?$$

$$\textbf{father}(x,z) \wedge \textbf{father}(z,\textit{Ermis})?$$

$$\textbf{father}(x,w) \wedge \textbf{father}(w,z) \wedge \textbf{father}(z,\textit{Ermis})?$$

$$\ldots$$

The answer to the original query on **ancestor** is the union of the answers to the infinite set of queries on **father**. Intuitively, this says that, to find someone's ancestors, one has to find their father(s), their grandfathers, their great-grandfathers, etc. □

Recursion may be present in a deductive database even if no Horn clause in the IDB is recursive. This happens if there is an interaction among the Horn clauses, with relations appearing on both the consequents of some of them and the qualifications of some others. This form of recursion is called *mutual recursion*, whereas the case of a single recursive Horn clause is called *immediate recursion*. Except for a very small part of this dissertation, we concentrate on studying immediate recursion only.

Example 1.5: The following set of Horn clauses gives rise to mutual recursion:

$$\textbf{likes\_reading}(x) \rightarrow \textbf{good\_student}(x)$$

$$\textbf{good\_student}(x) \rightarrow \textbf{likes\_reading}(x).$$

To the contrary, the following recursive Horn clause gives rise to immediate recursion:

$$\textbf{ancestor}(x,z) \wedge \textbf{father}(z,y) \rightarrow \textbf{ancestor}(x,y).$$ □

Recursive Horn clauses increase the power of database systems considerably. "Transfinite queries" are now expressible in finite form. The user can now ask queries, which even though are simple, conventional systems cannot answer. Some examples of such questions follow:

What are the ancestors of Mike?
What are the subparts of an airplane wing?
What is the shortest path between A and B?
.....

Notice that all the above questions can be seen as a form of generalized transitive closure of some (not necessarily binary) relation.

Despite its power, recursion has not been considered important in a business data processing environment. That is the reason why it was not introduced right from the early days of relational database systems. With the increasing need of combining artificial intelligence and database techniques, which imply the introduction of various forms of rules in database systems, recursion is now recognized as an essential characteristic of future systems. Therefore, being able to efficiently process queries in such an environment becomes critical.

1.4. Overview of Dissertation

The goal of this dissertation is to study processing techniques for queries on relations that are defined recursively. Unless otherwise mentioned, we concentrate on immediate recursion that arises from a range-restricted, function-free, linear, recursive Horn clause. Chapter 2 introduces an algebraic model for the study of recursion. Horn clauses are expressed by equations in some particular algebraic structure. Thus,

answering queries is accomplished by solving equations. In Chapter 3, a special kind of recursive Horn clauses is identified, namely recursive Horn clauses that are equivalent to a finite set of nonrecursive ones. For a restricted class of recursive Horn clauses, necessary and sufficient conditions are given for a Horn clause in the class to have this property. Unfortunately, most recursive Horn clauses do not enjoy this property. Hence, in Chapter 4, some general algorithms for answering queries involving recursively defined relations are proposed. With respect to performance, the new algorithms are compared against some traditional ones by analytical as well as experimental means, and their effectiveness is demonstrated in a large number of cases. However, which algorithm performs best depends on the database contents. This justifies the need for optimization algorithms. In Chapter 5, two such optimization algorithms are devised, one based on simulated annealing and another based on dynamic programming. Finally, the conclusions are reviewed and directions for future work are outlined in Chapter 6.

CHAPTER 2

CLOSED SEMIRING OF RELATIONAL OPERATORS

So far, in a database environment, recursion has been studied under the formalism of relational calculus, which is a subset of first order logic; the exceptions are few [Zani85]. A possible explanation is that in conventional database systems, where recursion is not allowed, relational algebra has not proved to be a useful representation for optimizing database commands. Recent results, however, indicate that, in the presence of recursion, relational algebra offers several advantages over relational calculus [Ioan86]. The subject of this chapter is the development of such an algebraic framework for the study of recursion.

2.1. Closed Semirings

Before investigating recursion from an algebraic viewpoint, some definitions from algebra are needed. Most of this subsection is taken from [Aho74].

Definition 2.1: A *closed semiring* is a system $(S,+,*,0,1)$, where S is a set of elements, and $+$ and $*$ are binary operators on S, satisfying the following properties:

1. $(S,+,0)$ is a *monoid*, i.e. it is closed under $+$ (if $a, b \in S$ then $a + b \in S$), $+$ is associative (if $a, b, c \in S$ then $(a + b) + c = a + (b + c)$) and 0 is an identity (if $a \in S$ then $a + 0 = 0 + a = a$). Likewise $(S,*,1)$ is a monoid. Finally, 0 is assumed to be an *annihilator* (if $a \in S$ then $a*0 = 0*a = 0$).

2. The operation + is commutative (if $a, b \in S$ then $a + b = b + a$) and idempotent (if $a \in S$ then $a + a = a$).

3. The operation $*$ distributes over + (if $a, b, c \in S$ then $a*(b + c) = a*b + a*c$ and $(b + c)*a = b*a + c*a$).

4. If $\{a_1, a_2, \cdots, a_i, \cdots\}$ is a countable set of elements in S then the sum $\sum_{i=1}^{\infty} a_i$ exists and is unique and an element of S. Moreover, associativity, commutativity, and idempotence apply to infinite as well as finite sums.

5. The operation $*$ distributes over countably infinite sums as well as finite ones.

Example 2.1: The following system taken from [Aho74] can be easily seen to be a closed semiring: ({*false,true*},OR,AND,false,true). □

Inductively, the powers of an element a of a closed semiring may be defined as:

$$a^0 = 1, \quad a^n = a^{n-1}*a = a*a^{n-1}, \quad \forall \ n \geq 0.$$

In the following a closed semiring with set S is represented by E_S.

Definition 2.2: Consider a closed semiring E_S and an element $a \in S$. The *transitive closure* of a, denoted by a^*, is defined as:

$$a^* = \sum_{i=0}^{\infty} a^k.$$

Notice that property 4 of closed semirings guarantees the existence of a^* in S. Also notice the similarity between the definition of a closed semiring and a *path algebra* [Carr79,Rose86]. The only difference is that a path algebra does not necessarily satisfy properties 4 and 5.

2.2. Relational Algebra

Consider a fixed (infinite) set of constants *CONST*. A database D is a vector $D = (CONST_D, R_1, ..., R_n)$, where $CONST_D \subseteq CONST$ is a ***finite*** set, and for each $1 \leq i \leq n$, $R_i \subseteq CONST_D^{a_i}$ is a relation of *arity* a_i. Each element of R_i is called a *tuple*.

Relational algebra has been introduced by Codd [Codd70] to formally describe the operations performed on relations in a database system. This dissertation focuses on a subset of the operators of the original proposal of relational algebra. We are interested in the set $S = \{X, \sigma_q, \pi_p\}$ of relational operators, where

X: Cross product of relations.

σ_q: Selection of tuples in the relation satisfying some constraint q of the form "column1 *op* column2", with $op \in \{=, \geq, >, \leq, <\}$.

π_p: Projection of the relation on a subset of its columns, specified by p.

The restriction to these operators is only for convenience. We claim that all the other interesting relational operators can be expressed using the ones in S. *Natural* join, denoted by $\bowtie$, is equal to a cross product followed by an equality selection and projection of the joined column. A constant c in the constraint of a selection can be replaced by the name given to the column of the set $\{c\}$. Intersection is equal to $\bowtie$ also. For convenience only, $\bowtie$ and selection with constants are occasionally used as abbreviations of their equivalent fully expanded forms. There are only two relational operators from the original proposal that are not incorporated in this analysis, namely division of relations and set-difference. The significance of the exclusion of the latter from S becomes clear shortly. Finally, arithmetic functions, or for that matter

functions in general, are not allowed in the operators.

2.3. Closed Semiring of Relational Operators

Consider a database $D = (CONST_D, R_1, ..., R_n)$ and the set S of primitive relational operators for D. Each element of S can be seen as a *unary* operator applied on some relation. This is obvious for selection and projection. For cross product, one of the operand relations is designated as a parameter of the operator, so that the operator is applied on the other relation alone. In this sense, an operator $A \in S$ is a mapping $A : CONST_D^a \rightarrow CONST_D^b$. $CONST_D^a$ is the *domain* and $CONST_D^b$ is the *range* of A. Operators with the same domain are called *domain-compatible* and operators with the same range are called *range-compatible*. Likewise, if the domain of an operator A is the same as the range of an operator B, then A is called *dr-compatible* to B (dr for domain-range).

Consider $S = \{X, \sigma_q, \pi_p\}$, the set of primitive relational operators for database D. Using S as the basis set, the system $E_R = (R, +, *, 0, 1)$ is defined as follows:

R The set of elements is defined as follows:

- If $A \in S \cup \{0, 1, \omega^\dagger\}$ then $A \in R$.
- If $A, B \in R$ then $(A + B) \in R$
- If $A, B \in R$ then $(A * B) \in R$
- Nothing else is an element of R.

---

† The operator ω can be thought of as the error operator. Applied on any nonempty relation it results in the relation $\{ERROR\}$, whereas $\omega\emptyset = \emptyset$.

+ For A, B domain- and range-compatible operators in R, addition is defined by $(A + B)\mathbf{P} = A\mathbf{P} \cup B\mathbf{P}$. Otherwise, $A + B = \omega$.

* For A, B operators in R with A dr-compatible to B, multiplication is defined by $(A * B)\mathbf{P} = A(B\mathbf{P})$. Otherwise, $A * B = \omega$.

0 The operator 0 can be applied on any relation and returns always the empty relation: $0\mathbf{P} = \emptyset$

1 The operator 1 can be applied on any relation and returns the relation unchanged: $1\mathbf{P} = P$

For notational convenience the multiplication symbol * is omitted. Whenever $AB\mathbf{P}$ is used, with $A, B \in R$ and $\mathbf{P}$ a relation, it actually represents $(A * B)\mathbf{P}$. Notice that, regarding the error operator ω, $\omega 1 = 1\omega = \omega$ and $\omega 0 = 0\omega = 0$.

Before proceeding in investigating the structure of E_R, some characteristic properties of the relational operators in R are identified.

Definition 2.3: A relational operator $A \in R$ is *linear* iff

(a) For all relations $\mathbf{P}$, $\mathbf{Q}$ in its domain, $A(\mathbf{P} \cup \mathbf{Q}) = A\mathbf{P} \cup A\mathbf{Q}$, and

(b) $A\emptyset = \emptyset$.

Proposition 2.1: If $A \in R$ then A is linear.

Proof: Consider an operator $A \in R$. The claim is proved by induction on k, the number of times addition and multiplication is applied on operators in $S \cup \{0,1,\omega\}$ to get A.

Basis: For $k = 0$, $A \in S \cup \{0,1,\omega\}$. It is simple to show that all these operators are linear.

Induction Step: Assume this is true for all operators formed using up to $k-1$ multiplications and additions. Let A be an operator that needs k such operations. The last operation is either addition or multiplication. So A has one of the following forms:

(i) $\mathrm{A} = \mathrm{B} + \mathrm{C} \Longrightarrow \mathrm{A}(\mathbf{P} \cup \mathbf{Q}) = (\mathrm{B} + \mathrm{C})(\mathbf{P} \cup \mathbf{Q})$

$\Longrightarrow A(\mathbf{P} \cup \mathbf{Q}) = B(\mathbf{P} \cup \mathbf{Q}) \cup C(\mathbf{P} \cup \mathbf{Q})$ definition of +

$\Longrightarrow A(\mathbf{P} \cup \mathbf{Q}) = (B\mathbf{P} \cup B\mathbf{Q}) \cup (C\mathbf{P} \cup C\mathbf{Q})$ induction hypothesis

$\Longrightarrow A(\mathbf{P} \cup \mathbf{Q}) = (B\mathbf{P} \cup C\mathbf{P}) \cup (B\mathbf{Q} \cup C\mathbf{Q})$ associativity of $\cup$

$\Longrightarrow A(\mathbf{P} \cup \mathbf{Q}) = (B + C)\mathbf{P} \cup (B + C)\mathbf{Q}$ definition of +

$\Longrightarrow A(\mathbf{P} \cup \mathbf{Q}) = A\mathbf{P} \cup A\mathbf{Q}.$

$A = B + C \Longrightarrow A\emptyset = (B + C)\emptyset$

$\Longrightarrow A\emptyset = B\emptyset \cup C\emptyset$ definition of +

$\Longrightarrow A\emptyset = \emptyset.$ induction hypothesis

(ii) $\mathrm{A} = \mathrm{B}\,\mathrm{C} \Longrightarrow \mathrm{A}(\mathbf{P} \cup \mathbf{Q}) = (\mathrm{B}\,\mathrm{C})(\mathbf{P} \cup \mathbf{Q})$

$\Longrightarrow A(\mathbf{P} \cup \mathbf{Q}) = B(C(\mathbf{P} \cup \mathbf{Q}))$ definition of *

$\Longrightarrow A(\mathbf{P} \cup \mathbf{Q}) = B(C\mathbf{P} \cup C\mathbf{Q})$ induction hypothesis

$\Longrightarrow A(\mathbf{P} \cup \mathbf{Q}) = BC\mathbf{P} \cup BC\mathbf{Q}$ induction hypothesis

$\Longrightarrow A(\mathbf{P} \cup \mathbf{Q}) = A\mathbf{P} \cup A\mathbf{Q}.$

$A = BC \Longrightarrow A\emptyset = (BC)\emptyset$

$\Longrightarrow A\emptyset = B(C\emptyset)$ definition of *

$$\Longrightarrow A\emptyset = B\emptyset \qquad \text{induction hypothesis}$$

$$\Longrightarrow A\emptyset = \emptyset. \qquad \text{induction hypothesis}$$

In both cases A is proved to be linear. □

Hereafter, unless otherwise mentioned, the term relational operator refers to a linear relational operator.

Definition 2.4: A relational operator is *monotone* iff for all relations **P**, **Q** in its domain $\mathbf{P} \subseteq \mathbf{Q} \Longrightarrow A\mathbf{P} \subseteq A\mathbf{Q}$.

Proposition 2.2: If a relational operator is linear then it is monotone.

Proof:

$$\mathbf{P} \subseteq \mathbf{Q} \Longrightarrow \mathbf{P} \cup \Delta\mathbf{Q} = \mathbf{Q} \Longrightarrow A(\mathbf{P} \cup \Delta\mathbf{Q}) = A\mathbf{Q}$$

$$\Longrightarrow A\mathbf{P} \cup A\Delta\mathbf{Q} = A\mathbf{Q} \Longrightarrow A\mathbf{P} \subseteq A\mathbf{Q}.$$ □

With respect to the exclusion of set-difference from the set of primitive relational operators S, notice that set-difference is neither linear nor monotone. The system of linear relational operators E_R is characterized algebraically by the following theorem:

Theorem 2.1: The system $E_R = (R,+,*,0,1)$ defined as above is a closed semiring.

Proof: The proof of properties 1, 2 and 3 of Definition 2.1 follows directly from the definitions of + and *. For point 4, it is sufficient to show that any countable sum of operators in R is equal to a finite sum. Let $A = A_1 + A_2 + \cdots$ be a countable sum of operators in R. Without loss of generality, assume that each A_i is a product of primitive operators in S. Otherwise, it can be brought into this form by applying the distributivity property of * over +. In addition, the following observations restrict

the form of A_i further:

- If $A_i = B_1 \cdots B_k 1 B_{k+1} \cdots B_n$, then $A = A_1 + \cdots + A'_i + \cdots$, with $A'_i = B_1 \cdots B_k B_{k+1} \cdots B_n$.
- If $A_i = B_1 \cdots B_k 0 B_{k+1} \cdots B_n$, then $A = A_1 + \cdots + A_{i-1} + A_{i+1} + \cdots$.
- If $A_i = B_1 \cdots B_k \omega B_{k+1} \cdots B_n$, and for all $1 \leq j \leq n$ $B_j \neq 0$, then $A = \omega$.
- If for some i, j, A_i and A_j are not domain- or not range-compatible, then $A = \omega$.

Justified by the above, the assumption is that all A_i are formed out of projections, selections, and cross products, and they all are domain- and range-compatible with each other. It is trivial to show that the projections and selections in A_i can be moved to the left and the cross products can be multiplied together to produce a single cross product with a large relation. Hence, each operator A_i takes the form $A_i = \pi\sigma^{(1)}\sigma^{(2)} \cdots \sigma^{(k_i)}(\mathbf{Q}_i X)$. The projection or the selections or the cross product may be missing. The above represents the most general form of A_i.

Let $A = A_1 + A_2 + \cdots$, with $A_i : CONST_D^a \rightarrow CONST_D^b$. With respect to the form of A_i, we distinguish two cases:

(a) There is no cross product in A_i. In this case all the terms of A_i (a projection or some selections or both) are domain-compatible. They are applied on relations in the domain of A. From the definition of selection in Section 2.2, there is only a finite number of conceivable selections applicable on relations of a specific domain. This is true for projections as well. Hence, there is only a finite number of operators formed without a cross product.

(b) A_i is a cross product, possibly multiplied to the left by some selections and a projection. Let $A_i = \pi\sigma^{(1)}\sigma^{(2)} \cdots \sigma^{(k_i)}(\mathbf{Q}_i X)$. Let $\mathbf{Q}_i \subseteq CONST_D^{c_i}$. The arity c_i of $\mathbf{Q}_i$ can be arbitrarily large. We claim that any such relation can be replaced by another one of arity less than N, for some finite N. Without loss of generality, assume that

$$c_i \leq b + 2k_i. \tag{2.1}$$

This is because we need at most b columns for the output relation, and exactly two columns for each selection; any other columns of $\mathbf{Q}_i$ can be projected out. We show that k_i is bounded by a fixed number. Consider selection $\sigma_{q_j}^{(j)}$, with q_j of the form "column1 *op* column2", $op \in \{=, \geq, >, \leq, <\}$. Partition the selections into three sets:

$S_1 = \{\sigma_{q_j}^{(j)} : q_j = (\text{col1 } op \text{ col2}), \text{ col1, col2 from the input relation}\}$

$S_2 = \{\sigma_{q_j}^{(j)} : q_j = (\text{col1 } op \text{ col2}), \text{ col1, col2 from } \mathbf{Q}_i\}$

$S_3 = \{\sigma_{q_j}^{(j)} : q_j = (\text{col1 } op \text{ col2}), \text{ col1 from the input relation, col2 from } \mathbf{Q}_i\}$.

Each one of S_1, S_2, and S_3 can be reduced to a finite set:

(i) There are only $5a^2$ distinct selections on relations of arity a. Hence, S_1 is finite.

(ii) Consider $\sigma_{q_j}^{(j)} \in S_2$. Any such selection on columns of $\mathbf{Q}_i$ can be preprocessed. Replace $\mathbf{Q}_i$ with $\mathbf{Q}'_i \subseteq \mathbf{Q}_i$, so that every tuple $t \in \mathbf{Q}'_i$ satisfies q_j, and remove $\sigma_{q_j}^{(j)}$ from A_i. Hence, S_2 can be reduced to the empty set.

(iii) If the cardinality of S_3 is greater than $5a$, it must contain two selections $\sigma_{q_j}^{(j)}$ and $\sigma_{q_{j'}}^{(j')}$, such that $q_j = (\text{col1 } op \text{ col2})$ and $q_{j'} = (\text{col1 } op \text{ col3})$, with col1 in the input relation, and col2 and col3 in $\mathbf{Q}_i$.

- If *op* is $=$, the two selections can be replaced by σ_q, where q=(col2 op col3), which belongs to S_1.
- If *op* is $\geq$ or $>$, each tuple of $\mathbf{Q}_i$ can be replace by one, whose value in col2 is the maximum of the values in col2 and col3. The selection $\sigma_{q_j}^{(j')}$ can then be removed.
- If *op* is $\leq$ or $<$, the above can be applied with the minimum of the values in col2 and col3.

In all cases the number of selections has been reduced by one. Hence, the operator can be transformed so that the cardinality of S_3 is no greater than $5a$.

The arguments in (i), (ii), and (iii) above imply that A_i can be transformed so that $k_i \leq 5a^2 + 5a$. This together with (2.1) implies that $c_i \leq N$, for some fixed N. Since $\mathbf{Q}_i \subseteq CONST_D^{c_i}$, there is only a finite number of such relations. With the same argument as in case (a), there is only a finite number of selections and projections applied on relations produced by applying $(\mathbf{Q}_i X)$ on a relation of arity a. Hence, there is a finite number of such operators all together.

From cases (a) and (b) it is implied that every countable sum of operators is equal to a finite sum. Therefore it is a member of R. Points 4 and 5 of Definition 2.1 are direct consequences of this. □

Had the set-difference operator been included in R, multiplication would not always distribute over addition (if d is set-difference and A, B two other operators then $d(A + B) \neq dA + dB$), and E_R would not be a closed semiring.

Since E_R is a closed semiring, the definitions for the n-th power and the transitive closure of a relational operator A that is dr-compatible to itself follow directly:

$$A^n = A * A * \cdots * A \quad (n \text{ times}),$$

with $A^0 = 1$ and

$$A^* = \sum_{k=0}^{\infty} A^k.$$

For any specific database D, A^* is a finite sum. This follows directly from the proof of Theorem 2.1. However, notice that the number of powers of A needed depends on the particular relations that are parameters of A and is different for each database D. This is completely different from the notion of bounded recursion, which is addressed in Chapter 3. In bounded recursion, A^* is equal to the same finite sum for all databases.

An interesting question is whether E_R is a richer system than simply a closed semiring. Posing the question more specifically, whether E_R is a ring, the answer is unfortunately negative. In order for E_R to be a ring, every relational operator must have an additive inverse, i.e. for every $A \in R$ another operator $B \in R$ must exist such that $A + B = 0$.

Proposition 2.3: The system $E_R = (R,+,*,0,1)$ defined above on the relational operators R is not a ring.

Proof: It suffices to find one operator in R that lacks an additive inverse. The multiplicative identity 1 serves this purpose. Assume that there exists an operator -1 such that $1 + (-1) = 0$. Then, for any nonempty relation **P**,

$$(1 + (-1))\mathbf{P} = \emptyset \Longrightarrow \mathbf{P} \cup (-1)\mathbf{P} = \emptyset,$$

which is a contradiction since $\mathbf{P}$ was taken to be nonempty. □

Equality of relational operators is naturally defined through set equality as

$$A = B \Longleftrightarrow \forall\, \mathbf{P},\ A\mathbf{P} = B\mathbf{P}.$$

Moreover, since + is associative, idempotent, and commutative, system E_R may be enriched in structure in another dimension. A partial order can be defined on R using set inclusion:

$$A \leq B \Longleftrightarrow \forall\, P,\ A\mathbf{P} \subseteq B\mathbf{P}.$$

Evidently, with respect to this ordering, 0 is the least element in R. Strict partial order $<$ as well as the duals $>$ and $\geq$ are defined analogously.

Proposition 2.4: Let $A, B, C, D \in R$ be relational operators appropriately compatible. With respect to multiplication and addition the partial order defined above enjoys the following properties:

(a) $A \leq A + B$

(b) $A \leq B \Longleftrightarrow A + B = B$

(c) $A \leq B \Longrightarrow A + C \leq B + C$

(d) $A \leq B$, $C \leq D$ and A, B are monotone $\Longrightarrow A\ C \leq B\ D$

Proof: The proofs of these properties are straightforward and are omitted. For (d), Propositions 2.1 and 2.2 assure that all relational operators under consideration are monotone. □

2.4. Algebraic Formulation of Recursion

Consider a range restricted, function-free, linear recursive Horn clause

$$\mathbf{P}(\underline{x}^{(0)}) \wedge \mathbf{Q}_1(\underline{x}^{(1)}) \wedge \cdots \wedge \mathbf{Q}_k(\underline{x}^{(k)}) \rightarrow \mathbf{P}(\underline{x}^{(k+1)}), \tag{2.2}$$

where for each i, $\underline{x}^{(i)}$ is a subset of some fixed set of variables $(x_1,x_2,...,x_n)$. Such a Horn clause can be expressed in relational terms as follows: Let $\mathbf{P}$, $\{\mathbf{Q}_i\}$ be relations and $f(\mathbf{P}, \{\mathbf{Q}_i\})$ a function with values that are relations over the same columns as $\mathbf{P}$. Then, (2.2) takes on the form

$$f(\mathbf{P},\{\mathbf{Q}_i\}) \cup \mathbf{P},$$

or equivalently

$$\mathbf{P} \cup f(\mathbf{P},\{\mathbf{Q}_i\}) = \mathbf{P}.$$

The problem of recursive inference can now be stated in relational form as follows: Given fixed relations $\mathbf{Q},\mathbf{Q}_1,...,\mathbf{Q}_k$ and function f, find $\mathbf{P}$ such that

(a) $\mathbf{P} \cup f(\mathbf{P},\{\mathbf{Q}_i\}) = \mathbf{P}$

(b) $\mathbf{Q} \subseteq \mathbf{P}$

(c) $\mathbf{P}$ is minimal with respect to (a) and (b), i.e. if $\mathbf{P}'$ satisfies (a) and (b) then $\mathbf{P} \subseteq \mathbf{P}'$.

Relation $\mathbf{Q}$ corresponds to a nonrecursive Horn clause of the form:

$$\mathbf{Q}(\underline{x}) \rightarrow \mathbf{P}(\underline{x}).$$

Conditions (a), (b) and (c) are equivalent to:

(a) $\mathbf{Q} \cup f(\mathbf{P},\{\mathbf{Q}_i\}) = \mathbf{P}$

(b) $\mathbf{P}$ is minimal with respect to (a), i.e. $\mathbf{P}'$ satisfying (a) implies $\mathbf{P} \subseteq \mathbf{P}'$.

This last set is the one we try to satisfy.

Consistent with the previous analysis, the function $f(\mathbf{P},\{\mathbf{Q}_i\})$, having $\{\mathbf{Q}_i\}$ as parameters and $\mathbf{P}$ as input, can be thought of as a linear relational operator (a product of some primitive operators, since f corresponds to a single Horn clause) applied on the recursive relation $\mathbf{P}$ to produce another relation union-compatible with it. Since the operator corresponds to a Horn clause, it is an element of R and therefore is linear and monotone [Chan76]. Hence, the established algebraic framework can be used to define the problem of immediate recursion. Consider a recursive Horn clause that corresponds to a linear operator A, so that

$$A\mathbf{P} \subseteq \mathbf{P}.$$

Consider some constant relation $\mathbf{Q}$ which is either stored or produced by some other nonrecursive Horn clause, so that

$$\mathbf{Q} \subseteq \mathbf{P}.$$

The relation defined by the Horn clause is the minimal solution to the equation

$$\mathbf{P} = A\mathbf{P} \cup \mathbf{Q}. \tag{2.3}$$

Presumably, the solution is a function of $\mathbf{Q}$. Hence, $\mathbf{P}$ is written as $\mathbf{P} = B\mathbf{Q}$, and the problem becomes one of finding the operator B. For the time being no assumption is made about B. In particular we do not presuppose that $B \in R$. Manipulation of (2.3) results in the elimination of $\mathbf{Q}$, so that the equation contains operators only. In this pure operator form the recursion problem can be restated as follows: Given operator A, find B satisfying:

$$\text{(a)}\quad 1+AB = B$$
$$\text{(b)}\quad 1+AC = C \Longrightarrow B \leq C \qquad (2.4)$$

Condition (b) guarantees that the solution found is the minimal one satisfying equation (a).

Theorem 2.2: Consider equation (2.4a) with restriction (2.4b). Its solution is A^*.

Proof: It has already been mentioned that $A \in R$, so it is linear and monotone. The system E_R is a closed semiring (Theorem 2.1). Thus, A^* exists and is unique for any A. First, A^* is a solution of (2.4a):

$$1 + AA^* = 1 + A(1 + A + \cdots) = 1 + A + A^2 + \cdots = A^*.$$

The second equality is due to the property that multiplication distributes over countable sums. Second, A^* is indeed the minimal solution (least fixpoint) of $1 + AB = B$. That is, for all operators B that satisfy (2.4a), $B \geq A^*$. This is shown by induction on the number of terms in $A^* = \sum_{k=0}^{\infty} A^k$.

Basis: For $n=0$, $\sum_{k=0}^{0} A^k = 1$, and from (2.4a) and Proposition 2.4b $B \geq 1$.

Induction Step: Assume that $B \geq \sum_{k=0}^{n} A^k$ for some $n \geq 0$. Then

$$B \geq \sum_{k=0}^{n} A^k \Longrightarrow AB \geq A \sum_{k=0}^{n} A^k \qquad \text{Proposition 2.4d}$$

$$\Longrightarrow 1 + AB \geq 1 + A \sum_{k=0}^{n} A^k \qquad \text{Proposition 2.4c}$$

$$\Longrightarrow \quad 1 + AB \geq \sum_{k=0}^{n+1} A^k \qquad \text{closed semiring properties}$$

$$\Longrightarrow \quad B \geq \sum_{k=0}^{n+1} A^k. \qquad \text{from (2.4a)}$$

So, for all $n \geq 0$, $B \geq \sum_{k=0}^{n} A^k$. Since the sequence (of the partial sums) is upwards bounded by B and is monotone, its limit A^* is also bounded by B. Hence, for any B satisfying $B = 1 + AB$, $B \geq A^*$. This implies that A^* is the least such operator, i.e. it is the least fixpoint of (2.4a). □

Theorem 2.2 is originally due to Tarski [Tars55] and in the database context was first examined by Aho and Ullman [Aho79a]. It is the first time though that the solution takes on an explicit form within an algebraic structure like the closed semiring E_R. The implications of the manipulative power thus afforded on the implementation of A^* are discussed in Chapters 4 and 5.

2.5. Transitive Closure as a Pseudo-Inverse

The lack of additive and multiplicative inverse impairs our ability to manipulate algebraic expressions of relational operators. In this regard, the transitive closure A^* of a relational operator A plays a useful role. It has been shown that $A^* = 1 + AA^*$ (A^* is a solution of (2.4a)). Blindly solving for A^*, as if $+$ and $*$ were real number addition and multiplication in the equation, gives

$$(1-A)A^* = 1, \quad A^* = (1-A)^{-1}. \tag{2.5}$$

Neither $1-B$ nor B^{-1} is well defined for every relational operator B. Nonetheless, for any algebraic expression that can be simplified using $(1-A)^{-1}$, A^* can be used in its place.

Example 2.2: Consider the equation

$$C = AC + B.$$

Solving for C and replacing $(1-A)^{-1}$ with A^* gives $C=A^*B$ as its solution. □

Example 2.3: For a more interesting example consider the equation

$$A + CDA = B + CA + DA.$$

Blindly solving for A, gives

$$(1 - C - D + CD)A = B,$$

which can then be transformed by the appropriate factorization to

$$(1 - C)(1 - D)A = B.$$

Multiplying both sides with the "multiplicative inverses" of $(1 - C)$ and $(1 - D)$ and applying (2.5) whenever possible yields the solution $A = D^*C^*B$. The fact that the above constitutes a solution to the original equation is easily verified by a simple substitution of D^*C^*B for A in it. This solution for A is hardly expected from the original equation. □

2.6. Mutual Linear Recursion

Until this point we have concentrated on immediate recursion. However, using the algebraic framework analyzed above, the same ideas can be applied to the cases where mutual recursion exists as well.

Definition 2.5: Consider a set of Horn clauses, and let $\{\mathbf{P}_1,\mathbf{P}_2,...,\mathbf{P}_n\}$ be the relations in the consequents of its elements. The set of Horn clauses is called *linear* iff each Horn clause has at most one of $\{\mathbf{P}_1,\mathbf{P}_2,...,\mathbf{P}_n\}$ in its qualification.

Example 2.4: The following system of mutually recursive Horn clauses is linear:

$$\mathbf{Q}(x,z) \wedge \mathbf{T}(z,y) \rightarrow \mathbf{P}(x,y)$$
$$\mathbf{P}(y,x) \rightarrow \mathbf{P}(x,y)$$
$$\mathbf{P}(z,x) \wedge \mathbf{S}(z,z,y) \rightarrow \mathbf{Q}(x,y)$$
$$\mathbf{R}(x,y) \rightarrow \mathbf{Q}(x,y).$$

To the contrary, the next one is not, because of the presence of both **P** and **Q** in the qualification of the first Horn clause.

$$\mathbf{P}(w,z) \wedge \mathbf{Q}(x,z) \wedge \mathbf{T}(z,y) \rightarrow \mathbf{P}(x,y)$$
$$\mathbf{P}(y,x) \rightarrow \mathbf{P}(x,y)$$
$$\mathbf{P}(z,x) \wedge \mathbf{S}(z,z,y) \rightarrow \mathbf{Q}(x,y)$$
$$\mathbf{R}(x,y) \rightarrow \mathbf{Q}(x,y).$$ □

Notice that this definition of linear is different (more restrictive) from the one given in [Banc86b]. That definition includes systems that are not linear. In particular, it includes systems that can be broken into smaller linear systems. These can be solved in such an order that the relations produced by one become parameters to the next one. We believe that a more precise term for such a system is *piecewise linear,* and we use the term linear according to Definition 2.5 (see also [Cosm86]).

Consider a linear system of mutually recursive Horn clauses defining relations $\{\mathbf{P}_1,\mathbf{P}_2,...,\mathbf{P}_n\}$. Each Horn clause is represented algebraically using a linear operator in R. This way a system of n equations is generated with n unknown variables $\{\mathbf{P}_1,\mathbf{P}_2,...,\mathbf{P}_n\}$ and is solved as an ordinary linear system. The interactions between the Horn clauses can be arbitrarily complex as long as the resulting system is linear. The possibility of immediate recursion is not excluded either. The system produced is

general enough to give the solution for the relations concerned.

Example 2.5: Consider the most general case of two relations **P** and **Q**, that are defined by both immediately recursive and mutually recursive Horn clauses in a linear way. Using linear operators from R, the situation is represented by the following linear system:

$$\mathbf{P}_1 = A\mathbf{P}_1 \cup B\mathbf{P}_2 \cup \mathbf{Q}_1$$

$$\mathbf{P}_2 = C\mathbf{P}_1 \cup D\mathbf{P}_2 \cup \mathbf{Q}_2.$$ □

We define the set $M_n(R)$ of nXn, $n \geq 1$, matrices whose entries belong to the set of linear relational operators R. Notice that for any such matrix, all the operators in a column are domain-compatible and all the operators in a row are range-compatible. Consider $\mathbf{A} = [A_{ij}]$, $\mathbf{B} = [B_{ij}]$, two matrices in $M_n(R)$. If $\forall\ 1 \leq i,j \leq n$, A_{ij} and B_{ij} are domain-compatible then **A, B** are called *domain-compatible*. Likewise, if $\forall\ 1 \leq i,j \leq n$, A_{ij} and B_{ij} are range-compatible then **A, B** are called *range-compatible*. Finally, if $\forall\ 1 \leq i,j,k \leq n$ A_{ik} is dr-compatible to B_{kj}, then **A** is called *dr-compatible* to **B**. Using $M_n(R)$ the system $E_{M_n(R)} = (M_n(R),+,*,\mathbf{0},\mathbf{1})$ is defined as follows:

$+$ If **A, B** are two matrices in $M_n(R)$ then addition is defined by $\mathbf{A} + \mathbf{B} = [A_{ij} + B_{ij}]$

$*$ If **A, B** are two matrices in $M_n(R)$ then multiplication is defined by $\mathbf{A} * \mathbf{B} = [\sum_{k=1}^{n} A_{ik}B_{kj}]$

0 The matrix **0** has all its elements equal to 0.

1 The matrix **1** has all its elements equal to 0, except the ones on the principle diagonal, which are equal to 1.

Similarly to the situation for simple operators the multiplication symbol $*$ is omitted.

Proposition 2.5: System $E_{M_n(R)} = (M_n(R),+,*,\mathbf{0},\mathbf{1})$ is a closed semiring.

Proof: It follows directly from the fact that E_R is a closed semiring and the algebraic property that matrices over a closed semiring form a closed semiring. □

Powers of matrices are defined as

$$\mathbf{A}^m = \mathbf{A} * \mathbf{A} * \cdots * \mathbf{A} \quad (m \text{ times}),$$

with $\mathbf{A}^0 = \mathbf{1}$ and the transitive closure of a matrix as

$$\mathbf{A}^* = \sum_{k=0}^{\infty} \mathbf{A}^k.$$

The finiteness of $\mathbf{A}^*$ for every specific database follows in the same way as for simple relational operators.

Consider a linear system of equations like the one of Example 2.5. Using elements of $M_n(R)$ it can be written in matrix form as

$$\underline{\mathbf{P}} = \mathbf{A}\underline{\mathbf{P}} \cup \underline{\mathbf{Q}}, \tag{2.6}$$

with $\underline{\mathbf{P}}$ the vector of unknown relations and $\underline{\mathbf{Q}}$ the vector of stored relations. Since both E_R and $E_{M_n(R)}$ are closed semirings, the minimal solution to (2.6) can be found in exactly the same way as that of (2.3) and is $\underline{\mathbf{P}} = \mathbf{A}^* \underline{\mathbf{Q}}$.

Example 2.6: Consider the linear system of Example 2.5. Written in matrix form it is equal to

$$\begin{bmatrix}\mathbf{P}_1\\ \mathbf{P}_2\end{bmatrix} = \begin{bmatrix}A & B\\ C & D\end{bmatrix}\begin{bmatrix}\mathbf{P}_1\\ \mathbf{P}_2\end{bmatrix} \cup \begin{bmatrix}\mathbf{Q}_1\\ \mathbf{Q}_2\end{bmatrix}.$$

Solving the system we get that

$$\begin{bmatrix}\mathbf{P}_1\\ \mathbf{P}_2\end{bmatrix} = \begin{bmatrix}A & B\\ C & D\end{bmatrix}^* \begin{bmatrix}\mathbf{Q}_1\\ \mathbf{Q}_2\end{bmatrix}.$$

The individual solutions for $\mathbf{P}_1$ and $\mathbf{P}_2$ are:

$$\mathbf{P}_1 = (A + BD^*C)^*(BD^*\mathbf{Q}_2 \cup \mathbf{Q}_1)$$

$$\mathbf{P}_2 = (D + CA^*B)^*(CA^*\mathbf{Q}_1 \cup \mathbf{Q}_2).$$ □

The importance of the algebraic formulation of the problem should be emphasized at this point. Until now, few people have dealt with mutual recursion in its full generality. The methods proposed for finding the relations defined by a highly recursive system of Horn clauses seem to be complicated and in cases incomplete [Hens84, Viei86]. The power of the algebraic tools gives the solution for arbitrarily complex linear systems almost for free, in a concise way. The following more complex example illustrates the effectiveness of this approach.

Example 2.7: Consider the case of three mutually recursive relations, with no immediate recursion for any of them. The linear system representing the situation is

$$\begin{bmatrix}\mathbf{P}_1\\ \mathbf{P}_2\\ \mathbf{P}_3\end{bmatrix} = \begin{bmatrix}0 & A_{21} & A_{31}\\ A_{12} & 0 & A_{32}\\ A_{13} & A_{23} & 0\end{bmatrix}\begin{bmatrix}\mathbf{P}_1\\ \mathbf{P}_2\\ \mathbf{P}_3\end{bmatrix} \cup \begin{bmatrix}\mathbf{Q}_1\\ \mathbf{Q}_2\\ \mathbf{Q}_3\end{bmatrix}.$$

Solving for $\mathbf{P}_1$, for example, the solution is

$$\mathbf{P}_1 = [\,(A_{12} + A_{13}A_{32})(A_{23}A_{32})^*A_{21} + (A_{13} + A_{12}A_{23})(A_{32}A_{23})^*A_{31}\,]^*\mathbf{Q}, \quad (2.7)$$

where relation Q is equal to

$$\mathbf{Q} = \mathbf{Q}_1 \cup (A_{12} + A_{13}A_{32})(A_{23}A_{32})^*\mathbf{Q}_2 \cup (A_{13} + A_{12}A_{23})(A_{32}A_{23})^*\mathbf{Q}_3.$$

The expression may be long, nevertheless it gives a complete solution of the linear system for $\mathbf{P}_1$. This was hard to find before, if at all possible. □

2.7. Regular Languages and Relational Operators

An interesting relationship exists between the algebraic view of recursion and regular expressions and finite state automata. Consider a finite alphabet Σ and take the system $E_{Po(\Sigma^*)} = (Po(\Sigma^*),\cup,\cdot,\emptyset,\lambda)$ on the set of regular expressions on Σ, $Po(\Sigma^*)$†. It is known that $E_{Po(\Sigma^*)}$ is a closed semiring [Aho74]. Hence, because of the isomorphism of the structure between E_R and $E_{Po(\Sigma^*)}$ a solution like (2.7) can be viewed as a regular expression. The isomorphism maps multiplication of operators to concatenation of strings, addition to union, transitive closure to Kleene star, 1 to the empty string λ, and 0 to $\emptyset$. This suggests a different way of finding the solution to a linear system of Horn clauses by constructing a Finite State Diagram (FSD) and finding the corresponding regular expression. The FSD is constructed from the given linear system as follows: There are two states in the diagram for each one of the n unknown relations; call them S_i and F_i, $1 \leq i \leq n$. For each i, $1 \leq i \leq n$, there is the transition $S_i \rightarrow F_i$ on input $\mathbf{Q}_i$. Also, for each i,j , $1 \leq i,j \leq n$, there is a transition $S_i \rightarrow S_j$ on input A_{ij} (the element of $\mathbf{A}$ in the i-th row and j-th column). The final states are F_i, $1 \leq i \leq n$. The solution for $\mathbf{P}_i$, $1 \leq i \leq n$ is the regular expression

† Given a set S its powerset, i.e. the set of its subsets , is represented by $Po(S)$.

corresponding to the FSD having S_i as its initial state.

Example 2.8: Consider the linear system of Example 2.7 and construct the corresponding FSD as described above. It is shown in Figure 2.1.

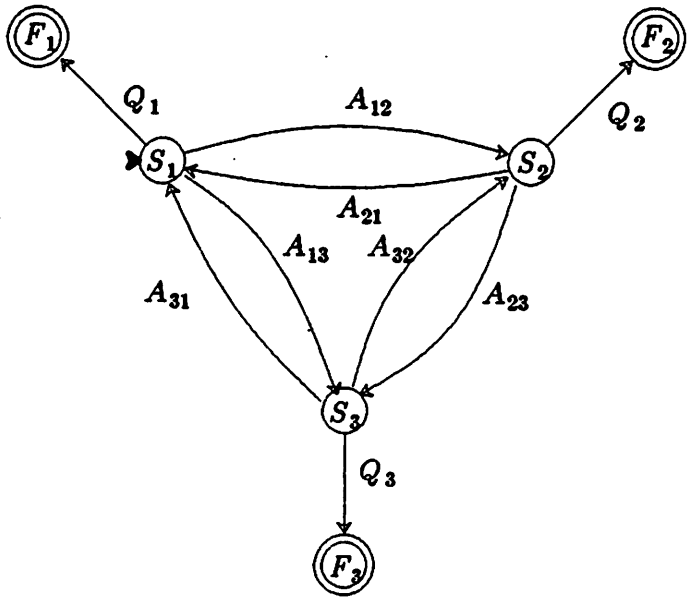

Figure 2.1. FSD for the solution of 3-relation mutual recursion.

The solution for $\mathbf{P}_1$ is given in (2.7). It is straightforward to verify that (2.7) is the regular expression corresponding to the diagram of Figure 2.1. □

2.8. Nonlinear Relational Operators

The power of the algebraic approach is once more demonstrated in the case of nonlinear recursion. In particular, the issue of the possible equivalence of nonlinear and linear recursive Horn clauses is addressed for the case of immediate recursion. Consider the canonical example for a system of one linear recursive and one nonrecursive Horn clause:

$$\textbf{ancestor}(x,z) \wedge \textbf{father}(z,y) \rightarrow \textbf{ancestor}(x,y)$$

$$\textbf{father}(x,y) \rightarrow \textbf{ancestor}(x,y).$$

It is clear that **ancestor** is equal to the transitive closure of **father**. Observe that the above system can be equivalently expressed using a nonlinear Horn clause as:

$$\begin{aligned} &\textbf{ancestor}(x,z) \wedge \textbf{ancestor}(z,y) \rightarrow \textbf{ancestor}(x,y) \\ &\textbf{father}(x,y) \rightarrow \textbf{ancestor}(x,y). \end{aligned} \tag{2.8}$$

The second clause in (2.8) has **ancestor** appearing twice in the qualification, hence it is nonlinear. The question that arises is under what conditions a nonlinear recursive Horn clause is equivalent to a linear one.

We concentrate on *quadratic* Horn clauses, where the recursive relation appears exactly twice in the qualification, like in (2.8) above. The assumptions about the Horn clause being range restricted and function-free are still in effect. Using the algebraic framework developed in Section 2.3 each system of one quadratic recursive and one nonrecursive Horn clause is written as:

$$A[\mathbf{P}]\mathbf{P} \cup \mathbf{Q} = \mathbf{P}. \tag{2.9}$$

The only difference between (2.9) and (2.3) is that the operator applied on $\mathbf{P}$ depends on $\mathbf{P}$. The notation $A[\mathbf{P}]$ is used to indicate that relation $\mathbf{P}$ is a parameter to the operator A. For each quadratic recursive Horn clause there is a choice of which of the two appearances of the recursive relation in the qualification is the parameter to the operator. Instead of (2.9), we may take

$$B[\mathbf{P}]\mathbf{P} \cup \mathbf{Q} = \mathbf{P}, \tag{2.10}$$

with $A[\mathbf{P}]\mathbf{P}' = B[\mathbf{P}']\mathbf{P}$ for all relations $\mathbf{P}, \mathbf{P}'$. Notice that both $A[\mathbf{P}]$ and $B[\mathbf{P}]$ are linear operators in R. Also notice that they are linear in terms of their parameter

also, i.e. $A[\mathbf{P} \cup \mathbf{Q}] = A[\mathbf{P}] + A[\mathbf{Q}]$ and $A[\emptyset] = 0$.

The solution to (2.9) (or (2.10)) can be found by iteration [Aho79a]:

$\mathbf{P}_0 = Q$

$\mathbf{P}_k = A[\mathbf{P}_{k-1}]\mathbf{P}_{k-1} \cup \mathbf{Q}$.

Due to the finiteness of the database and the absence of function from the operators, the iteration is guaranteed to terminate at some point, where $\mathbf{P}_{N+1} = \mathbf{P}_N$ for some N. The solution for $\mathbf{P}$ is $\mathbf{P} = \mathbf{P}_N$.

Definition 2.6: Consider a quadratic Horn clause corresponding to the operator $A[\mathbf{P}]$. It is equivalent to a linear one, if in equation (2.9) $A[P]$ can be replaced by $A[\mathbf{Q}]$

$$A[\mathbf{Q}]\mathbf{P} \cup \mathbf{Q} = \mathbf{P}. \tag{2.11}$$

The minimal solution of (2.11) is known to be $\mathbf{P} = A[\mathbf{Q}]^*\mathbf{Q}$, so (2.9) is equivalent to a linear equation if and only if $A[\mathbf{Q}]^*\mathbf{Q}$ satisfies it.

Theorem 2.3: Consider equation (2.9). If, for all relations $\mathbf{P}$, $A[A[\mathbf{P}]\mathbf{P}] = A[\mathbf{P}]A[\mathbf{P}] = A^2[\mathbf{P}]$, then (2.9) is equivalent to a linear equation $A[\mathbf{Q}]\mathbf{P} \cup \mathbf{Q} = \mathbf{P}$.

Proof: Let $f(\mathbf{P})$ denote the function $A[\mathbf{P}]\mathbf{P} \cup \mathbf{Q}$.

$$f(\mathbf{P}) = A[\mathbf{P}]\mathbf{P} \cup \mathbf{Q}. \tag{2.12}$$

The power $f^n(\mathbf{P})$ is inductively defined as $f^0(\mathbf{P}) = \mathbf{P}$ and $f^n(\mathbf{P}) = f^{n-1}(f(\mathbf{P})) = f(f^{n-1}(\mathbf{P}))$. In this notation, the solution of (2.9) takes on the form $\mathbf{P} = f^N(\mathbf{Q})$ for some integer N. It suffices to prove that $f^N(\mathbf{Q})$ can take the form

$$f^N(\mathbf{Q}) = \bigcup \left[\prod_{\substack{l,m \in \{0,L\} \\ k \geq 0}} A^k[f^l(\mathbf{Q})]\, f^m(\mathbf{Q}) \right], \tag{2.13}$$

for any $0 \leq L \leq N$. This is accomplished by a backward induction on L, the maximum power of $f(.)$ appearing in (2.13).

Basis: For $L = N$ the statement clearly holds, since $\mathbf{P} = f^N(\mathbf{Q})$.

Induction step: Assume that the statement is true for L. It is proved for $L-1$. Take an arbitrary term $t = A^k[f^l(\mathbf{Q})]f^m(\mathbf{Q})$ from the union of products. We distinguish four cases:

$l=m=0$: The term is left unchanged: $A^k[\mathbf{Q}]\mathbf{Q}$.

$l=0$, $m=L$: The term t is transformed according to the definition of f (2.12):

$$\begin{aligned} t &= A^k[\mathbf{Q}]\,(A[f^{L-1}(\mathbf{Q})]\,f^{L-1}(\mathbf{Q}) \cup \mathbf{Q}) \\ &= A^k[\mathbf{Q}]A[f^{L-1}(\mathbf{Q})]\,f^{L-1}(\mathbf{Q}) \;\cup\; A^k[\mathbf{Q}]\mathbf{Q}. \end{aligned}$$

linearity of $A^k[\mathbf{Q}]$

$l=L$, $m=0$: Again, the term t is written according to the definition of f (2.12) as:

$$\begin{aligned} t &= A^k[A[f^{L-1}(\mathbf{Q})]\,f^{L-1}(\mathbf{Q}) \cup \mathbf{Q}]\mathbf{Q} \\ &= \left[A[A[f^{L-1}(\mathbf{Q})]\,f^{L-1}(\mathbf{Q})] + A[\mathbf{Q}]\right]^k \mathbf{Q} \end{aligned}$$

linearity of A[.] with respect to its parameter

$$\begin{aligned} &= \left[A^2[f^{L-1}(\mathbf{Q})] + A[\mathbf{Q}]\right]^k \mathbf{Q} \\ &= \bigcup \left[\prod_{l' \in \{0,\, k \geq 0\}^{L-1}} A^{k'}[f^{l'}(\mathbf{Q})]\right] \mathbf{Q}. \end{aligned}$$

hypothesis of the theorem

$l=m=L$: In this case the transformations of the two previous cases are combined.

In all four cases each term of the union of products is transformed and written as

$$f^N(\mathbf{Q}) = \bigcup \prod_{l,m \in \{0,\, k \geq 0\}^{L-1}} A^k[f^l(\mathbf{Q})]f^m(\mathbf{Q}).$$

This concludes the induction step, which proves (2.13). Therefore, choosing $L=0$, $f^N(\mathbf{Q})$ takes on the form

$$f^N(\mathbf{Q}) = \bigcup \left[\prod_{\substack{l=m=0 \\ k\geq 0}} A^k[f^l(\mathbf{Q})]\, f^m(\mathbf{Q}) \right] = \bigcup \left[\prod_{k\geq 0} A^k[\mathbf{Q}]\mathbf{Q} \right]$$

$$= \bigcup_{k\geq 0} \left[A^k[\mathbf{Q}]\mathbf{Q} \right] = \left[\sum_{k\geq 0} A^k[\mathbf{Q}] \right] \mathbf{Q} = A^*[\mathbf{Q}]\mathbf{Q}. \qquad \square$$

In the Horn clause formulation, the condition of Theorem 2.3 is stated as follows. Consider a quadratic recursive Horn clause h defining relation **P**. Get a second instance h' of the Horn clause, with its "input" relation replaced by a new arbitrary relation symbol **Q**. The two Horn clauses h and h' are *resolved* [Robi65] in two ways (see Figure 2.2): One by resolving the consequent of h with the sole instance of **P** in the qualification of h' and another by resolving the consequent of h' with the "input" instance of **P** in the qualification of h. If the two resolvents are the same, the Horn clause is equivalent to a linear one. The latter is formed by replacing the "parameter" instance of **P** in the qualification of the former with the basis relation for **P**.

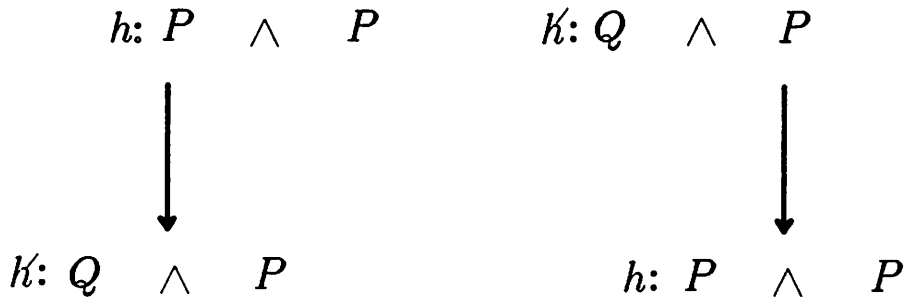

Figure 2.2. Resolving to test linear equivalence.

Example 2.9: Consider the **ancestor** example again for which it is known that it is equivalent to a linear Horn clause.

h : **ancestor**(x,z) ∧ **ancestor**(z,y) → **ancestor**(x,y).

Let the leftmost instance of **ancestor** be "input". Get h' by replacing it by another relation name, say **arbitrary**.

$$h': \quad \mathbf{arbitrary}(x,z) \wedge \mathbf{ancestor}(z,y) \rightarrow \mathbf{ancestor}(x,y).$$

Performing the resolutions mentioned above we get

$$\mathbf{arbitrary}(x,z) \wedge \mathbf{ancestor}(z,z') \wedge \mathbf{ancestor}(z',y) \rightarrow \mathbf{ancestor}(x,y)$$

$$\mathbf{arbitrary}(x,z') \wedge \mathbf{ancestor}(z',z) \wedge \mathbf{ancestor}(z,y) \rightarrow \mathbf{ancestor}(x,y).$$

The two Horn clauses are the same (up to renaming of variables). □

One peculiar related fact is that whether a quadratic Horn clause is equivalent to a linear one or not, does depend on which of (2.9), (2.10) is chosen to represent it. This can be shown by demonstrating a Horn clause, for which $A[A[\mathbf{P}]\mathbf{P}] = A^2[\mathbf{P}]$ holds but $B[B[\mathbf{P}]\mathbf{P}] = B^2[\mathbf{P}]$ does not.

Example 2.10: Consider the quadratic Horn clause

$$\mathbf{P}(x,z) \wedge \mathbf{P}(y,v) \rightarrow \mathbf{P}(x,y).$$

Let the leftmost occurrence of **P** in the qualification be "input", and construct the two operators $A[A[\mathbf{P}]\mathbf{P}]$ and $A^2[\mathbf{P}]$ for the corresponding operator A:

$$\mathbf{arbitrary}(x,z) \wedge \mathbf{P}(y,z') \wedge \mathbf{P}(v,v') \rightarrow \mathbf{P}(x,y)$$

$$\mathbf{arbitrary}(x,z'') \wedge \mathbf{P}(z,v'') \wedge \mathbf{P}(y,v) \rightarrow \mathbf{P}(x,y).$$

Clearly, the two Horn clauses are equal. To the contrary, the same procedure with the rightmost occurrence of **P** as "input" yields

$$\mathbf{P}(x,z) \wedge \mathbf{P}(y,z') \wedge \mathbf{arbitrary}(v,v') \rightarrow \mathbf{P}(x,y)$$

$$\mathbf{P}(x,z'') \wedge \mathbf{P}(z,v'') \wedge \mathbf{arbitrary}(y,v) \rightarrow \mathbf{P}(x,y).$$

These are two different Horn clauses. Therefore, a quadratic recursive Horn clause may be equivalent to a linear one with respect to one of the occurrences of the recursive relation in the qualification but not to the other. □

2.9. Summary

A significant subset of all the function-free linear relational operators have been embedded into a closed semiring. Within this algebraic structure, recursive inference by Horn clauses has been reduced to solving recursive equations. For a single linear Horn clause, the solution to the corresponding operator equation is equal to the transitive closure of the operator representing the Horn clause. This approach can be extended to multiple Horn clauses that are linearly mutual recursive. In that case, inference is reduced to solving linear systems of operator equations, in the same manner that immediate recursion is reduced to solving a single such equation. Finally, sufficient conditions are given for a nonlinear operator to be equivalent to a linear one.

CHAPTER 3

UNIFORMLY BOUNDED RECURSION

In Chapter 2 we argued that for every relational operator A, having relations $\{Q_1,Q_2,...,Q_m\}$ as parameters, $A^* = \sum_{k=0}^{N(\{Q_i\})} A^k = (1 + A)^{N(\{Q_i\})}$. That is, a finite number of terms is always sufficient for A^*. In general, the particular number depends on the database instance. However, there are some operators for which $A^* = \sum_{k=0}^{N} A^k = (1 + A)^N$, with N independent of the relation contents. A simple example is $A = 1$ the multiplicative identity. Clearly $1^* = 1$. In this chapter, we investigate the properties of such operators.

3.1. Algebraic Formulation

Definition 3.1: Consider a linear operator $A \in R$. If $A^* = (1 + A)^N$ for some N, then A is called *uniformly bounded.*

Another way of looking at the problem is the following: Consider the cyclic semigroup C_{1+A} generated by $(1 + A)$ [Clif61]. The elements of the semigroup are $(1 + A), (1 + A)^2,...,(1 + A)^k,....$ An operator A is bounded if C_{1+A} is finite.

Definition 3.2: Consider the closed semiring E_R and an operator $B \in R$.. If there exist natural numbers p and q, $p < q$, such that $B^p = B^q$, then B is called *torsion.*

Proposition 3.1: Consider an operator $A \in R$. If $1 + A$ is torsion then A is bounded.

Proof: Obvious. □

Definition 3.3: Consider a finite cyclic semigroup C_B, for which $B^p = B^q$, $p < q$, and p and q the smallest such numbers. Then p is called the *index*, $q-1$ the *order*, and $q-p$ the *period* of B and C_B [Clif61].

Some partial results in characterizing bounded operators are given below.

Proposition 3.2: The following conditions are equivalent:

(i) A is uniformly bounded with order N.

(ii) $A^{N+1} \leq (1+A)^N$.

(iii) $(1+A)^N(1+A)^N = (1+A)^N$, i.e. $(1+A)^N$ is idempotent.

Proof:

(ii) $\Longleftrightarrow$ (i):

Clearly, boundedness with order N is equivalent to $(1+A)^{N+1} = (1+A)^N$. Hence,

$$(1+A)^{N+1} = (1+A)^N \iff (1+A)^N + A^{N+1} = (1+A)^N$$
$$\iff A^{N+1} \leq (1+A)^N. \qquad \text{Proposition 2.4b}$$

(i) $\Longrightarrow$ (iii):

Again, the starting point is $(1+A)^{N+1} = (1+A)^N$.

$$(1+A)^{N+1} = (1+A)^N \implies (1+A)^{N+2} = (1+A)^{N+1}$$
$$\implies (1+A)^{N+2} = (1+A)^N.$$

Iterating, we get $(1+A)^{2N} = (1+A)^N(1+A)^N = (1+A)^N$.

(iii)$\Longrightarrow$(ii):

We know that $(1+A)^{2N} = \sum_{k=0}^{2N} A^k$. Hence, (iii) and Proposition 2.4b imply that

$$\sum_{k=0}^{2N} A^k = \sum_{k=0}^{N} A^k \implies A^{N+1} \leq (1+A)^N. \qquad \square$$

Notice that, for any uniformly bounded operator A, the period of C_{1+A} is 1, and the order and index coincide.

Proposition 3.3: Any one of the following conditions is sufficient for an operator A to be uniformly bounded with order N:

(i) For some m, $A^m \geq A^m A^m$.

(ii) For some m, $A^m \leq 1$.

(iii) For some m and for any initial relation $\mathbf{R}$, $A^m\mathbf{R} = \emptyset$ or $A^m\mathbf{R} = \mathbf{Q}$, where $\mathbf{Q}$ some constant relation (i.e. $A^m\mathbf{R}$ is in some sense independent of $\mathbf{R}$).

Proof:

(i) Assuming the given condition we can derive the following:

$$A^m \geq A^m A^m \implies \sum_{k=0}^{2m} A^k = \sum_{k=0}^{2m-1} A^k \qquad \text{Proposition 2.4b}$$

$$\implies (1+A)(1+A)^{2m-1} = (1+A)^{2m-1}.$$

The uniform boundedness of A is a direct consequence of the last formula.

(ii) Multiplying both sides of $A^m \leq 1$ by A^m yields $A^{2m} \leq A^m$, which is the condition of (i) above. Hence, A is uniformly bounded.

(iii) If $A^m\mathbf{R} = \emptyset$, then $A^m(A^m\mathbf{R}) = \emptyset$ also. If $A^m\mathbf{R} = \mathbf{Q}$ then $A^m(A^m\mathbf{R}) = A^m\mathbf{Q}$ is equal to either $\mathbf{Q}$ or $\emptyset$. Nevertheless, in both cases $A^m(A^m\mathbf{R}) \subseteq A^m\mathbf{R}$ for all $\mathbf{R}$, which by definition implies $A^mA^m \leq A^m$. Hence, condition (i) can be applied to give the uniform boundedness of A. □

3.2. Simple Recursive Horn Clauses

Even though finite semigroups have been algebraically characterized within different settings [Mand77,Hash79,Pele84], to our knowledge finite semigroups of linear operators in R have not. The characterization results presented here are based on the Horn clause formulation of recursion.

Definition 3.4: Two variables x, y *appear under the same relation* in a Horn clause if there is an atomic formula $\mathbf{P}(...,x,...,y,...)$ appearing in the Horn clause, where $\mathbf{P}$ is a relation symbol.

Definition 3.5: A variable is called *distinguished* if it appears under the recursive relation in the consequent of a Horn clause. Otherwise it is called *nondistinguished*.

We restrict our attention to recursive Horn clauses that satisfy the following conditions:

R1 The recursive Horn clause is *linear*.

R2 There are *no function symbols* in the Horn clause.

R3 There are *no constant symbols* in the Horn clause.

R4 There are *no repeated* variables in the consequent.

R5 *No* subsequence of distinguished variables in the consequent is a *permutation* of the corresponding subsequence of the variables under the recursive relation in the antecedent.

The motivation behind restriction R1 is simplicity. Moreover, most of the recursive Horn clauses in a real world system are expected to be linear. Function symbols in a recursive Horn clause may lead to infinite relations. Situations like that are not easily handled in a database environment, if at all. Restriction R2 is imposed to avoid them. The last three restrictions are imposed for the sole purpose of getting a uniform result. It is the goal of our future research to remove them, thereby generalizing our results.

Finally, we assume that there are no equalities in the Horn clause. Any equality may be removed by replacing one of its variables with the other, wherever it appears in the Horn clause. It is clear that the new Horn clause is equivalent to the initial one.

Definition 3.6: A recursive Horn clause is called *simple* if it satisfies conditions R1 through R5 and does not contain any equality symbol.

3.3. Horn Clause Formulation

Consider a simple recursive Horn clause:

$$\mathbf{P}(x_1,x_2,\ldots,x_m) \wedge \beta \rightarrow \mathbf{P}(y_1,y_2,\ldots,y_m). \tag{3.1}$$

Subformula β is a conjunction of atomic formulas, with relations other than $\mathbf{P}$. The following infinite sequence of nonrecursive Horn clauses is equivalent to (3.1):

$$\mathbf{P}_0(x_1,x_2,\ldots,x_m) \wedge \beta^{(0)} \rightarrow \mathbf{P}_1(y_1,y_2,\ldots,y_m)$$
$$\mathbf{P}_0(x_1^{(1)},x_2^{(1)},\ldots,x_m^{(1)}) \wedge \beta^{(1)} \wedge \beta^{(0)} \rightarrow \mathbf{P}_2(y_1,y_2,\ldots,y_m)$$
$$\mathbf{P}_0(x_1^{(2)},x_2^{(2)},\ldots,x_m^{(2)}) \wedge \beta^{(2)} \wedge \beta^{(1)} \wedge \beta^{(0)} \rightarrow \mathbf{P}_3(y_1,y_2,\ldots,y_m)$$
$$\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots$$

In the above, $\beta^{(i)} = \beta[s_i]$ and $\mathbf{P}_0(x_1^{(i)},x_2^{(i)},\ldots,x_m^{(i)}) = \mathbf{P}_0(x_1,x_2,\ldots,x_m)[s_i]$, for some substitution s_i of the variables in (3.1). Note that s_0 substitutes each variable for

itself. The i-th Horn clause above is called the *(i-1)-th expansion* of (3.1). So, the first one of these Horn clauses is the 0-th expansion. Each one of these expansions is applied on $\mathbf{P}_0$, the initial contents of $\mathbf{P}$. $\mathbf{P}$ is equal to $\bigcup_{i=0}^{\infty} \mathbf{P}_i$. The i-th expansion of a recursive Horn clause α is denoted by α_i. Whenever it creates no confusion, the same is used to denote the recursive Horn clause produced by removing the subscripts from $\mathbf{P}$ in α_i.

In a database environment all the relations are finite. Furthermore, only function-free recursive Horn clauses are considered. Hence, even though a recursive Horn clause is equivalent to an infinite sequence of nonrecursive Horn clauses, the latter stop producing new tuples for $\mathbf{P}$ after some point. The process terminates exactly when some N-th expansion of (3.1) fails to produce any new tuples for the first time. In general, N depends on the database contents. Our goal is to identify and characterize simple recursive Horn clauses, for which this is not true, i.e. the number of expansions needed to materialize the relation defined is independent of the database contents.

The problem can be addressed in two frameworks. In the first, $\mathbf{P}_0$ is produced by a given nonrecursive Horn clause. In the second, $\mathbf{P}_0$ is stored in the database. The question in the first case is one of *boundedness*, whereas in the second is one of *uniform boundedness*. We address the question of uniform boundedness only.

Definition 3.7: A simple recursive Horn clause α is called *uniformly bounded* if, for some N, α is equivalent to $\alpha_0, \alpha_1, \ldots, \alpha_{N-1}$.

Definition 3.8: Let a uniformly bounded simple recursive Horn clause α be equivalent to its first N expansions, $\alpha_0, \alpha_1, \ldots, \alpha_{N-1}$. The smallest such N is called the *order* of α.

Example 3.1: Consider the following simple recursive Horn clause α:

$$\alpha: \quad \textbf{reachable}(x) \wedge \textbf{edge}(x,y) \rightarrow \textbf{reachable}(y).$$

If **edge** represents the edges of a directed graph, then **reachable** denotes the nodes of the graph reachable from the ones originally contained in it. In general, the number of times α has to be applied to get the materialization of **reachable** is not known. It depends on the initial contents of **edge** (the graph) and **reachable.** □

Example 3.2: As another example of a simple recursive Horn clause, consider β:

$$\beta: \quad \mathbf{P}(z) \wedge \mathbf{NP\_c}(z) \wedge \mathbf{NP\_c}(y) \rightarrow \mathbf{P}(y).$$

If **NP_c** is the set of NP-complete problems and **P** is the set of problems evaluated in polynomial time, then β states the well known theorem that if one NP-complete problem is in **P**, then all are [Lewi81]. Clearly, one application of β is enough to materialize **P**, regardless of the initial contents of the relations **P** and **NP_c**. If there is one tuple in **P** that joins with (that is, is equal to) some tuple in **NP_c**, then β produces for **P** all the tuples in **NP_c**. Any further applications of β fail to produce new tuples for **P**. So β, unlike α, is uniformly bounded with order equal to 1. □

Uniform boundedness of Horn clauses has been addressed in the past. Minker and Nicolas give a sufficient condition for a Horn clause to be uniformly bounded [Mink83]. In particular, they define a restricted class of Horn clauses, called *singular*, and show that any singular Horn clause is uniformly bounded. They do allow non-

linearity, but singular Horn clauses are restricted in the way relations share variables.

The problem has also been addressed under a tableau formulation [Sagi85]. Representing a Horn clause by a tableau, Sagiv gives necessary and sufficient conditions for a set of Horn clauses to be uniformly bounded. The restrictions imposed are that there exists only one relation symbol in the Horn clauses, and that the Horn clauses are *typed*, i.e. no variable appears in more than one column of the relation. Similar results have been given by Cosmadakis and Kanellakis also [Cosm86].

More recently, Naughton has addressed the same problem [Naug86]. The class of Horn clauses he considers is similar to the class of simple Horn clauses. He does not impose restriction R5, but he does not allow a nonrecursive relation to appear more than once in the antecedent either. Besides giving a necessary and sufficient condition for uniform boundedness for this class, he also addresses boundedness as well as the case with multiple recursive Horn clauses.

Some tools developed for the study of conjunctive queries [Chan76] and tableaux [Aho79b] are helpful in addressing the uniform boundedness problem of Horn clauses also.

Definition 3.9: A *valuation* θ is a function from variables to constants. Applying θ on an atomic formula $\mathbf{Q}(x_1,...,x_n)$ gives the tuple $<\theta(x_1),...,\theta(x_n)>$.

Definition 3.10: Consider two nonrecursive Horn clauses α and β. A *homomorphism* h: $\alpha \rightarrow \beta$ is a mapping from the variables of α into those of β, such that:

(i) If x, y are distinguished variables appearing in the same argument position in the consequent of α and β respectively, then $h(x)=y$.

(ii) If $\mathbf{Q}(x_1,...,x_n)$ appears in the antecedent of α, then $\mathbf{Q}(h(x_1),...,h(x_n))$ appears in the antecedent of β.

When it is well defined, composition of homomorphisms h_1 and h_2 is denoted by $h_1 \circ h_2$.

Definition 3.11: For two nonrecursive Horn clauses α and β, α is *more restrictive* than β, denoted $\alpha \leq_r \beta$, iff for any database instance the relation produced by α is a subset of that produced by β. Clearly, $\leq_r$ is a partial order.

Definition 3.12: For two nonrecursive Horn clauses α and β, α is *equivalent* to β, denoted $\alpha =_r \beta$, iff there exists an isomorphism $h : \alpha \rightarrow \beta$, that is a homomorphism that is one-to-one and onto.

The following lemma is a slight extension of a similar result on typed tableaux [Aho79b].

Lemma 3.1: For two nonrecursive Horn clauses α and β, $\alpha \leq_r \beta$ iff there exists a homomorphism $h : \beta \rightarrow \alpha$.

Proof: Let d_i (D_i), $0 \leq i \leq m$, be the distinguished variable in the i-th argument position in the consequent of α (β).

(*If*). Assume that there exists a homomorphism $h : \beta \rightarrow \alpha$. Let θ be a valuation on the variables of α. Consider a database instance such that relation $\mathbf{Q}$ contains the tuple $<\theta(x_1),...,\theta(x_n)>$ iff an atomic formula $\mathbf{Q}(x_1,...,x_n)$ appears in the antecedent of α. The composition of θ and h, $\theta' = \theta \circ h$, is a valuation on β. Property

(ii) of homomorphisms assures that for an atomic formula $\mathbf{Q}(y_1,...,y_n)$ in the antecedent of β, the tuple $<\theta'(y_1),...,\theta'(y_n)>\in\mathbf{Q}$. Property (i) of homomorphisms guarantees that $<\theta(d_i)> = <\theta'(D_i)>$, $1\leq i\leq m$. Thus, any tuple in the relation produced by α is in the one produced by β as well. So, $\alpha \leq_r \beta$.

(*Only if*). Consider a one-to-one valuation θ from the variables in α onto some set of constants C. Consider a database instance such that relation $\mathbf{Q}$ contains the tuple $<\theta(x_1),...,\theta(x_n)>$ iff an atomic formula $\mathbf{Q}(x_1,...,x_n)$ appears in the antecedent of α. Then the tuple $<\theta(d_1),...,\theta(d_n)>$ is in the relation produced by α. Since $\alpha \leq_r \beta$, it has to be in the relation produced by β as well. Thus, a valuation θ' from the variables of β into the set of constants C exists, such that $\theta'(D_i) = \theta(d_i)$, $0\leq i\leq m$, and for any atomic formula $\mathbf{Q}(y_1,...,y_n)$ in the antecedent of β, $<\theta'(y_1),...,\theta'(y_n)>\in\mathbf{Q}$. θ is one-to-one and onto, so its inverse θ^{-1} is defined. Taking the composition $h = \theta^{-1}\circ\theta'$, it is easy to verify that it is a homomorphism from the variables of β into the variables of α. □

Lemma 3.2. Let α_s and α_t, $0\leq s<t$, be two expansions of some simple recursive Horn clause α, such that $\alpha_t \leq_r \alpha_s$. Then, for all $k\geq 0$, $\alpha_{t+k} \leq_r \alpha_{s+k}$.

Proof: It is shown by induction on k. The basis case $k=0$ is given. Assume that $\alpha_{t+k-1} \leq_r \alpha_{s+k-1}$ is true. By Lemma 3.1, there exists a homomorphism $h : \alpha_{s+k-1} \rightarrow \alpha_{t+k-1}$. Consider α_{s+k} and α_{t+k}. From the way expansions are formed, it is clear that the new part in the antecedent of α_{s+k} is isomorphic to the new part in the antecedent of α_{t+k} (they are both equivalent to α_0). Let h_I be this isomorphism from the part of α_{s+k} onto the part of α_{t+k}. Consider the mapping

$h' : \alpha_{s+k} \rightarrow \alpha_{t+k}$ defined as follows:

$$h'(x) = \begin{cases} h(x) & \text{if } x \text{ appeared in } \alpha_{s+k-1} \\ h_I(x) & \text{if } x \text{ is new} \end{cases}$$

Clearly, h' is a homomorphism from α_{s+k} into α_{t+k}. By Lemma 3.1, $\alpha_{t+k} \leq_r \alpha_{s+k}$. □

Lemma 3.3: A simple recursive Horn clause α is uniformly bounded iff there exist n, N, $n<N$, such that $\alpha_N \leq_r \alpha_n$.

Proof: (*If*). By Lemma 3.2 and the transitivity of $\leq_r$, if $\alpha_N \leq_r \alpha_n$ then for all $k \geq N$ there exists some $n'<N$ such that $\alpha_k \leq_r \alpha_{n'}$. Hence, α is equivalent to its first N expansions. Its order is at most N.

(*Only if*). This part follows the proof of [Sagi80], and is similar to Lemma 3.1. Let α be equivalent to its N first expansions, α_0 through α_{N-1}. Assume that all of them share the same distinguished variables d_i, $1 \leq i \leq m$. Consider a one-to-one valuation θ from the variables of α_N onto some set of constants C. Consider a database instance such that relation **Q** contains the tuple $<\theta(x_1),...,\theta(x_l)>$ iff an atomic formula $\mathbf{Q}(x_1,...,x_l)$ appears in the antecedent of α_N. Then the tuple $<\theta(d_1),...,\theta(d_m)>$ is in the relation produced by α_N. Since α is equivalent to its first N expansions, it has to be in the relation produced by α_n, for some $n<N$, as well. Thus, a valuation θ' from the variables of α_n into the set of constants C exists, such that for any atomic formula $\mathbf{Q}(y_1,...,y_l)$ in the antecedent of α_n, $<\theta'(y_1),...,\theta'(y_l)> \in \mathbf{Q}$. θ is one-to-one and onto, so its inverse θ^{-1} is defined. It is easy to verify that the composition $h = \theta^{-1} \circ \theta'$ is a homomorphism from the variables of α_n into the variables of α_N. Hence, by Lemma 3.1, $\alpha_N \leq_r \alpha_n$. □

Consider a uniformly bounded simple recursive Horn clause of order N. By Lemma 3.3, there exists n, $n<N$, such that $\alpha_N \leq_r \alpha_n$. Whenever $\alpha_k \leq_r \alpha_l$ implies $(k-l) = c(N-n)$, for some integer $c \geq 0$, following the terminology of [Clif61], we define n to be the *index* and $N-n$ the *period* of α. Notice the difference in the definition of the same terms for relational operators.

3.4. The Model

The examples given in Section 3.3 indicate that the way in which the variables of a Horn clause are connected with each other through the relations, plays an important role in whether the Horn clause is uniformly bounded or not. In this section a graph model is developed for simple recursive Horn clauses. The form of the graph reflects the connection among the variables in the Horn clause.

Let α be a simple recursive Horn clause. It is modeled by a labeled, weighted, directed graph constructed as follows:

(i) There is a node in the graph for every variable in α.

(ii) If two variables x,y appear under some nonrecursive relation **Q** in α then an undirected edge $(x-y)$ is put in the graph between the corresponding two nodes x,y. The label of the edge is **Q** and its weight is 0.

(iii) If two variables x,y appear in the same argument position of the recursive relation **P** in the antecedent and the consequent respectively, then a directed edge $(x \rightarrow y)$ is put in the graph from node x to node y with weight 1 and its inverse edge $(y \rightarrow x)$ with weight -1. Each directed edge has label **P**.

The graph corresponding to a simple recursive Horn clause α is called the *α—graph*. The subgraph induced on the α-graph by the undirected edges defined in (ii) is called the *static α—graph*. The spanning subgraph of the α-graph with edge set its directed edges defined in (iii) is called the *dynamic α—graph*. The weight of a path (cycle) in the graph is the sum of the weights of the edges along the path (cycle). Regarding static edges, they can be traversed in both directions, as if there were two opposite directed edges.

Example 3.3: Consider the following simple recursive Horn clause:

$$\alpha: \quad \mathbf{P}(z,w) \wedge \mathbf{Q}(z,x) \wedge \mathbf{R}(w,u) \wedge \mathbf{S}(u,x,y) \rightarrow \mathbf{P}(x,y).$$

The α-graph is shown in Figure 3.1.

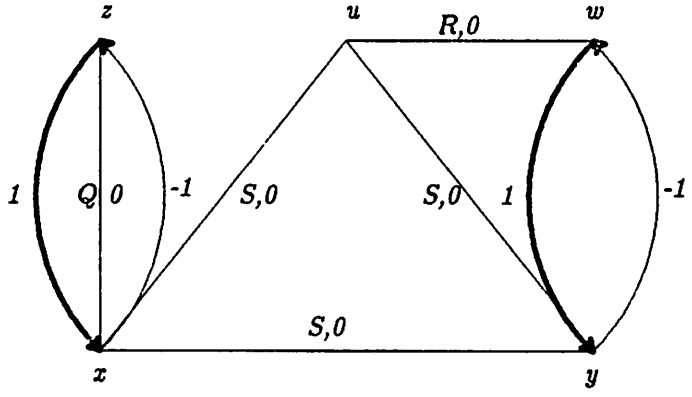

Figure 3.1. The α-graph.

□

In terms of the graph model, restrictions R4 and R5 together may be stated as

R4, R5 The dynamic graph (restricted on the positive edges) is a forest.

According to the definition of the graph model, there is a one-to-one correspondence between the positive and the negative dynamic edges. The positive ones alone are enough to carry all the information captured by the dynamic edges in the graph.

Hereafter, we refer to the dynamic graph as containing the positive edges only, the negative ones implicitly assumed only whenever the weight of a path is discussed. Likewise, in all the figures only the positive edges are drawn. Finally, since the weight of some edge is determined from whether it is static (weight zero) or dynamic (weight one), no weight is put on the edges.

3.5. Characterizing Uniform Boundedness

Uniform boundedness for simple recursive Horn clauses is characterized by the following theorem:

Theorem 3.1: A simple recursive Horn clause α is uniformly bounded iff the α-graph contains no cycle of nonzero weight. In that case the order of α is equal to the maximum path-weight in the α-graph.

Without loss of generality, we restrict our attention to Horn clauses that involve only binary nonrecursive relations. Any n-ary relation $\mathbf{Q}$, $n>2$, can be replaced by n binary relations $\mathbf{Q}_i$, constructed as follows. Each tuple of $\mathbf{Q}$ is assigned a unique identifier, called *tid* [Ston76]. Relation $\mathbf{Q}_i$ is created by combining tids with the corresponding values in the i-th column of $\mathbf{Q}$. Clearly, $\mathbf{Q}$ may be reconstructed from the $\mathbf{Q}_i$'s by joining all of them on the tid column. That generality is not lost when considering only binary nonrecursive relations is justified by Proposition 3.4:

Proposition 3.4: Let α be a simple recursive Horn clause and α' the Horn clause produced by replacing all n-ary, $n>2$, nonrecursive relations of α with binary relations according to the description above. The α-graph contains no nonzero weight cycle iff the α'-graph contains no nonzero weight cycle.

Proof: Any two variables that appear under some n-ary, $n>2$, relation in α are connected through a zero weight edge in the α-graph and through two zero weight edges in the α'-graph. The common node of the two edges is the one corresponding to the tid column. Hence, for any path in the α-graph there exists another one in the α'-graph with the same weight and vice-versa. □

Also, every (connected) component of the graph of some simple recursive Horn clause α, expands independently of the other. Hence, uniform boundedness of α is equivalent to uniform boundedness of all of its components. Lemma 3.4 formalizes the above.

Lemma 3.4: Let α be a simple recursive Horn clause and the α-graph consist of M connected components β^1 through β^M. If, for all $1 \leq i \leq M$, β^i is uniformly bounded then α is uniformly bounded as well. Moreover, if β^i is of order N_i and period P_i, α is of period $P = lcm\{P_i\}$ (the least common multiple of $\{P_i\}$) and order $N = \max_{1 \leq i \leq M}\{N_i - P_i\} + lcm\{P_i\}$.

Proof: Since each component expands without any interactions with the other, uniform boundedness of all of them implies uniform boundedness of the whole graph also. Since period is well defined for all β^i, it is well defined for α also. Let N be the order and P the period of α. This means that $\alpha_N \leq_r \alpha_{N-P}$, and N is the minimum such number. Moreover, this is the case for each β^i, that is $(\beta^i)_N \leq_r (\beta^i)_{N-P}$. This, Lemma 3.2, and the transitivity of $\leq_r$ imply that for all $1 \leq i \leq M$ there exist integers c_i, c'_i, and r_i, with $0 \leq c_i < c'_i$ and $0 \leq r_i \leq P_i - 1$, such that

$$(N_i - P_i) + c_i P_i + r_i = N - P \tag{3.2}$$

$$(N_i - P_i) + c'_i P_i + r_i = N. \tag{3.3}$$

Subtracting (3.2) from (3.3) yields $(c'_i - c_i)P_i = P$, which implies that P_i divides P, for all $1 \leq i \leq M$. Hence, $P = lcm\{P_i\}$. Since $r_i \geq 0$ and $c_i \geq 0$, (3.2) yields $N - P \geq N_i - P_i$, for all $1 \leq i \leq M$. Hence, $N - P = \max_{1 \leq i \leq} \{N_i - P_i\}$ or equivalently $N = \max_i \{N_i - P_i\} + lcm\{P_i\}$. Notice that this implies that the index of α is the maximum of the indices of the β^i's. □

For the proof of Theorem 3.1, a regular naming scheme for variables is established. Consider a simple recursive Horn clause α. Restrictions R4 and R5 denote that the dynamic α-graph is a forest; therefore, every connected component in the graph is a tree. For every node in such a tree there is a unique path from it to the root of the tree. Each variable is subscripted by the weight of this path, which is nonpositive. Hence, for a variable x_j,

$$x_j = \begin{cases} \text{nondistinguished variable} & \text{if } j = 0 \\ \text{distinguished variable} & \text{if } j < 0 \end{cases}. \tag{3.4}$$

Variables that belong to the same root-to-leaf path in the dynamic α-graph are denoted by the same symbol (with different subscripts).

Example 3.4: Figure 3.2 illustrates the established notation for the variables of some simple recursive Horn clause α.

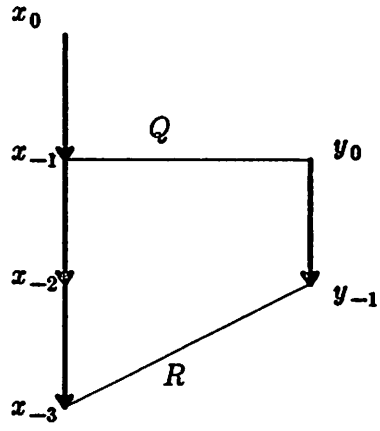

Figure 3.2. Example of variable naming.

The Horn clause corresponding to the figure is

$$\mathbf{P}(x_0,x_{-1},x_{-2},y_0) \wedge \mathbf{Q}(x_{-1},y_0) \wedge \mathbf{R}(y_{-1},x_{-3}) \rightarrow \mathbf{P}(x_{-1},x_{-2},x_{-3},y_{-1}). \qquad \square$$

In the first expansion of some Horn clause α, each distinguished variable is replaced by the corresponding variable appearing under the recursive relation in the antecedent. Due to the variable naming convention, this means that a distinguished variable x_{-k} is replaced by x_{-k+1}. In addition, some new variables are introduced to replace the nondistinguished variables. The convention is that each new variable has the name of the one it replaces with the superscript increased by 1. According to the above it may be inductively shown that in the n-th expansion x_i is replaced by x_{i+n}. So, the variable substitution for the expansions of α given in Section 3.3 is

$$s_n(x_i) = x_{i+n}, \tag{3.5}$$

meaning that x_{i+n} is substituted for x_i.

Lemma 3.5: Consider two variables x_k, x_l, with $k>l$. There is a path of weight $c>0$ from x_k to x_l in the dynamic α_n-graph, $n \geq 0$, iff the following hold:

(a) $k-l=c(n+1)$ the distance of the two variables in the dynamic α-graph is a multiple of $(n+1)$.

(b) $k, l \leq n$ both variables do appear in α_n.

Proof: (*If*). Let x_k,x_l be two variables satisfying (a) and (b). The conditions imply that

$$k-l = c(n+1) \Longrightarrow l = k-c(n+1) \leq n-c(n+1) = (1-c)n-c < 0.$$

Hence, (3.4) implies that x_l is a distinguished variable. The lemma is shown by induction on c.

Basis: For $c=1$, $k-l = n+1$. Since x_l is distinguished, there is an edge from x_{l+1} to it in the α-graph. The substitution in (3.5) implies that in the α_n-graph there is an edge from $x_{l+1+n} = x_k$ to x_l. Hence the two are connected with a path of weight 1.

Induction Step: Assume that the above is true for all integers less than c. Take variable x_j, with $j-l = (n+1)$. From the induction hypothesis there is an edge from x_j to x_l in the α_n-graph. Also, since $k-j = k-l-(n+1) = (c-1)(n+1)$, the induction hypothesis implies that there is a path of weight $(c-1)$ from x_k to x_j. Hence there is a path of weight c from x_k to x_l.

(*Only if*). Let x_k, x_l be two variables that are connected through a path of weight c in the α_n-graph. Substitution (3.5), implies that the first expansion variable x_k (x_l) appears is $\alpha_{\max\{k,0\}}$ $(\alpha_{\max\{l,0\}})$. Hence, $k, l \leq n$. Furthermore, (3.5) implies that the path of weight 1 from x_k leads to $x_{k-(n+1)}$. An easy induction shows that the path of weight c from x_k leads to $x_{k-c(n+1)}$. The last variable is equal to x_l. Hence,

$$k-c(n+1)=l \iff k-l = c(n+1). \qquad \square$$

Consider a path in the α_n-graph of some expansion of α. In general, this path corresponds to a *walk* in the α-graph (when traversing a walk, nodes and edges may be visited multiple times [Bond76]). Static edges met in a traversal of the path in the α_n-graph are met in the same order in a traversal of the walk in the α-graph. Likewise, a cycle in the α_n-graph corresponds to a cyclic walk in the α-graph (its end nodes coincide). The following lemma relates the path in the α_n-graph and the corresponding walk in the α-graph.

Lemma 3.6: Let x_k, x_l be two variables connected through a path of weight L_n in the α_n-graph. Let $x_{k'}$, $x_{l'}$ be the end variables of the corresponding walk in the α-graph, of weight L. Then the following holds:

$$(k-k') = (l-l') + (n+1)L_n - L. \tag{3.6}$$

Proof: Let $k=k_1$, $k'=k'_1$, $l=l_m$, and $l'=l'_m$. Figure 3.3 shows the path from x_k to x_l and the walk from $x_{k'}$ to $x_{l'}$ in the α_n- and the α-graph respectively.

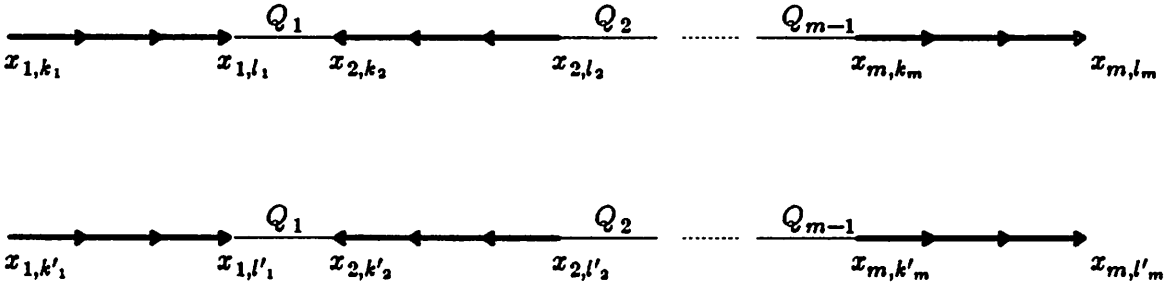

Figure 3.3. Path in the α_n-graph and corresponding walk in the α-graph.

By Lemma 3.5, each section in the dynamic α_n-graph is of length $(k_i-l_i) \,/\, (n+1)$. Hence,

$$L_n = \frac{1}{n+1} \sum_{i=1}^{m} (k_i - l_i). \tag{3.7}$$

Likewise, for the walk in the α-graph

$$L = \sum_{i=1}^{m} (k'_i - l'_i). \tag{3.8}$$

Consider $\mathbf{Q}_i(x_{i,l_i}, x_{i+1,k_{i+1}})$, $1 \leq i \leq m-1$, in the α_n-graph. Let n_i be the expansion that this was formed from $\mathbf{Q}_i(x_{i,l'_i}, x_{i+1,k'_{i+1}})$ in α. Substitution (3.5) implies that

$$\left.\begin{aligned} l_i &= l'_i + n_i \\ k_{i+1} &= k'_{i+1} + n_i \end{aligned}\right\} \Longrightarrow l_i - l'_i = k_{i+1} - k'_{i+1}. \tag{3.9}$$

Adding (3.9) for all $1 \leq i \leq m-1$ gives $\sum_{i=1}^{m-1} (l_i - l'_i) = \sum_{i=2}^{m} (k_i - k'_i)$. By (3.7) and (3.8), since $k = k_1$, $k' = k'_1$, $l = l_m$, and $l' = l'_m$, the above gets transformed into

$$(k - k') = (l - l') + (n+1)L_n - L. \qquad \square$$

Corollary 1: If there is a cycle of weight L_n in the α_n-graph, then there is a cyclic walk of weight $L = (n+1)L_n$ in the α-graph.

Proof: Assume that in Figure 3.3 there is one more static edge $\mathbf{Q}_m$ between x_{l_m} and x_{k_1} (see Figure 3.4).

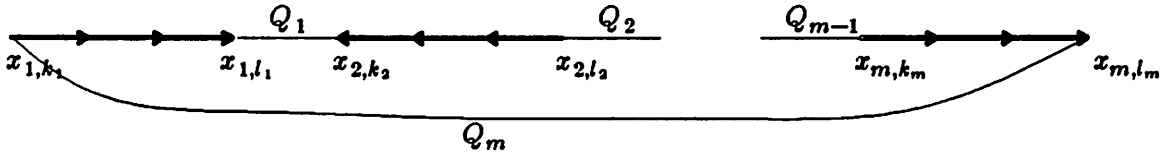

Figure 3.4. Cycle in the α_n-graph.

The corresponding edge in the α-graph is between $x_{l'_m}$ and $x_{k'_1}$, which creates a cycle in the α-graph as well. Like in (3.9), substitution (3.5) implies that $l_m - l'_m = k_1 - k'_1$.

Applying the above on (3.6) yields $L = (n+1)L_n$. □

3.5.1. Sufficiency of the Condition

Lemma 3.7: Let α be a simple recursive Horn clause such that the α-graph is connected and has no nonzero weight cycles. If N, $N \geq 1$, is the maximum path-weight in the α-graph, then $\alpha_N \leq_r \alpha_{N-1}$.

Proof: Consider α_{N-1}. Let $Q(x_m, x'_{m'})$ appear in its antecedent. Assume that α_n is the first expansion in which it appeared, corresponding to $Q(x_{-k}, x'_{-k'})$, $k, k' \geq 0$, in α. In this case, (3.5) implies that $n = k+m = k'+m'$. Consider a node z_0 in the α-graph whose distance from some other node in the graph is N, i.e. there is a path in the graph starting at that node whose weight is the maximum possible (see Figure 3.5).

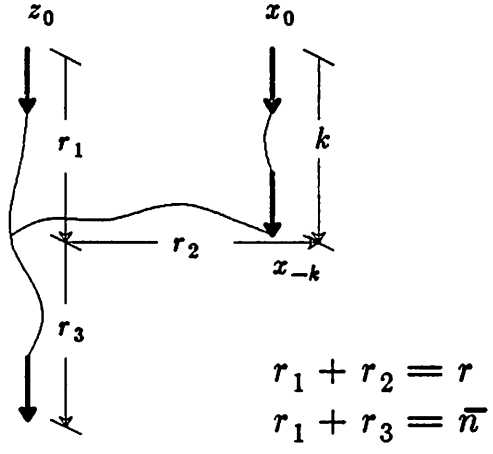

Figure 3.5. Distance between variables in the α-graph.

Let r be the distance of z_0 from x_{-k}. Consider the following mapping $h : \alpha_{N-1} \rightarrow \alpha_N$:

$$h(x_m) = \begin{cases} x_{m+1} & \forall\ m,\ r+1 \leq k+m \leq N-1 \\ x_m & \text{otherwise} \end{cases}.$$

If x_m does not map to itself then $m \geq 0$. Otherwise,

$$m < 0 \Longrightarrow k+m < k \Longrightarrow r < k \Longrightarrow r_1 + r_2 < k$$

$$\Longrightarrow r_1 + r_3 < k - r_2 + r_3 \Longrightarrow N < k - r_2 + r_3,$$

i.e. a path of length greater than N exists (see Figure 3.5). Hence, looking at (3.4) also, we can see that h is meaningful, i.e. it affects only nondistinguished variables. Since $k+m = k'+m'$, h affects x_m and $x'_{m'}$ the same way. Distinguished variables map to themselves. For every atomic formula $\mathbf{Q}(x_m, x'_{m'})$ in α_{N-1} there is another one $\mathbf{Q}(x_{m+1}, x'_{m'+1})$ in α_N, which appeared in α_{n+1} for the first time (see substitution (3.5)). The same is true for the recursive relation also. Hence, h is indeed a homomorphism from α_{N-1} into α_N. Lemma 3.1 implies that $\alpha_N \leq_r \alpha_{N-1}$. □

Theorem 3.2: Let α be a simple recursive Horn clause. If the α-graph contains no cycle of nonzero weight then α is uniformly bounded.

Proof: By Lemmas 3.3 and 3.7, each component of the α-graph is uniformly bounded. Lemma 3.4 implies that α is uniformly bounded as well. □

3.5.2. Order of Uniformly Bounded Simple Horn Clauses

Lemma 3.8: Let α be a simple recursive Horn clause. Consider α_s and α_t, $0 \leq s < t$, with $\alpha_t \leq_r \alpha_s$. Then the maximum path-weight in the dynamic α_s-graph is 1.

Proof: Lemma 3.1 implies that there exists a homomorphism $h : \alpha_s \rightarrow \alpha_t$. This induces a homomorphism h for the corresponding graphs also. Consider a single component G of the α_s-graph. Partition the nodes of G into two subgraphs H and H' as follows:

$$H = \{x \in G : h(x) \neq x\}$$

$$H' = \{x \in G : h(x) = x\}.$$

Distinguished variables map to themselves, so all heads of dynamic edges are in H'. Furthermore, restrictions R3 through R5 imply that no variable appears in the same argument position of the same relation in two different expansions (see s_n in (3.5)). Since the recursive relation appears exactly once in the antecedent of each expansion, no variable appearing under it in α_s can map to itself. Hence, all tails of dynamic edges are in H. The two points above imply that H is connected to H' as shown in Figure 3.6.

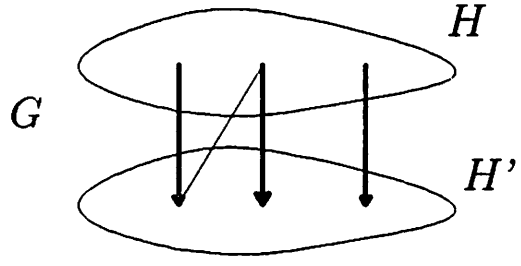

Figure 3.6. General form of the α_s-graph with $\alpha_t \leq_r \alpha_s$, for $t > s$.

Clearly, the maximum path-weight in the dynamic α_s-graph is 1. □

For the following analysis it is necessary to define the following family of functions f_n: $\mathbb{Z} \rightarrow \{0,1,...,n-1\}$, $\mathbb{Z}$ the set of integers, defined for all $n > 0$ as follows:

$$f_n(x) = (x \ mod \ n).$$

In the above, $(x \ mod \ n) = r$ if and only if n divides $(x-r)$ and $0 \leq r < n$.

Lemma 3.9: For all integers x, y and all positive integers n $(f_n(x) + y)$ - $f_n(x+y) = cn$ for some integer c.

Proof: Let $x = p_1 n + r_1$ and $y = p_2 n + r_2$, where $0 \leq r_1, r_2 < n$. Then,

$$(f_n(x) + y) - f_n(x+y) = (x \ mod \ n) + y - ((x+y) \ mod \ n) =$$

$$r_1 + p_2 n + r_2 - ((r_1+r_2) \ mod \ n) = (p_2+d)n, \text{ where } d \in \{0,1\}.$$

Therefore, $(f_n(x) + y) - f_n(x+y)$ is a multiple of n. □

Lemma 3.10: Let α be a simple recursive Horn clause. If the α-graph contains a path of weight L with ϵ static edges, then the α_n-graph contains a path of weight $\lfloor \frac{n+L}{n+1} \rfloor$ † with ϵ static edges.

Proof: Consider a path in the α-graph of weight L and $\epsilon = m-1$ (see Figure 3.7).

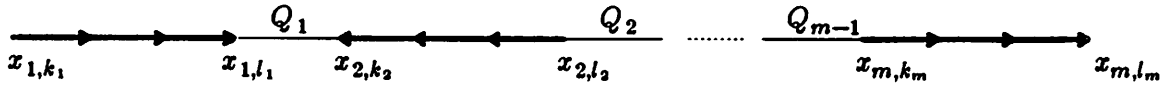

Figure 3.7. Typical path in the graph of some simple recursive Horn clause.

From Figure 3.7, the weight of the path from x_{1,k_1} to x_{m,l_m} is equal to

$$L = \sum_{i=1}^{m} (k_i - l_i). \tag{3.10}$$

From the way the expansions are formed, α_n contains $n+1$ instances of Q_i, $1 \leq i \leq m-1$, each one created at a different expansion. The one created at expansion

† By $\lfloor x \rfloor$ we denote the smallest integer greater than or equal to x.

r_i, $0 \leq r_i \leq n$, is of the form $\mathbf{Q}_i(x_{i,r_i+l_i}, x_{i+1,r_i+k_{i+1}})$. We claim that the combination of appropriately chosen instances of the $\mathbf{Q}_i$'s in α_n creates a path in the α_n-graph. Let σ_i be

$$\sigma_i = \sum_{j=1}^{i}(k_j - l_j), \quad \text{for} \quad 0 \leq i < m.$$

In Figure 3.7, σ_i denotes the distance of the variables appearing under $\mathbf{Q}_i$ from x_{1,k_1}. The $\mathbf{Q}_i$'s are chosen so that $r_i = f_{n+1}(n+\sigma_i)$. For every $\mathbf{Q}_{i-1}$, $\mathbf{Q}_i$ chosen as above the two variables $x_{i,r_{i-1}+k_i}$ and x_{i,r_i+l_i} are connected with a path in the dynamic α_n-graph. This is shown as follows. Lemma 3.9 implies that

$$(f_{n+1}(n+\sigma_{i-1}) + k_i) - (f_{n+1}(n+\sigma_i) + l_i) = c(n+1), \tag{3.11}$$

for some integer c. Furthermore, from the definition of f_{n+1} and $k_i, l_i \leq 0$

$$f_{n+1}(n+\sigma_{i-1}) + k_i < n+1 \quad \text{and} \quad f_{n+1}(n+\sigma_i) + l_i < n+1.$$

Hence, the conditions of Lemma 3.5 are satisfied. Therefore, the given variables are connected in the dynamic α_n-graph by a path of weight c (as given in (3.11)). Since this is true for all $1 \leq i \leq m$, the $\mathbf{Q}_i$'s chosen as above form a path in the α_n-graph with $\epsilon = m-1$ static edges.

Let L_n be the weight of the path. Using (3.11) for the weight of each individual subpath, L_n becomes equal to

$$L_n = \frac{1}{n+1} \sum_{i=1}^{m} \left[f_{n+1}(n+\sigma_{i-1}) + k_i - f_{n+1}(n+\sigma_i) - l_i \right] \iff$$

$$L_n = \frac{1}{n+1} \left[f_{n+1}(n+\sigma_0) - f_{n+1}(n+\sigma_m) + \sum_{i=1}^{m} (k_i - l_i) \right].$$

From (3.10), which gives the weight L of the original path, and the fact that $\sigma_0 = 0$ we get

$$L_n = \frac{1}{n+1}\Big[\, n + L - f_{n+1}(n+L) \,\Big] \iff L_n = \lfloor \frac{n+L}{n+1} \rfloor.$$

The last equivalence is an obvious implication of the definition of f_{n+1}. □

Theorem 3.3: Let α be a simple recursive Horn clause such that the α-graph contains no nonzero weight cycles. The order of α is equal to the maximum path-weight in the α-graph.

Proof: Suppose that the maximum path-weight in the α-graph is N. Theorem 3.2 shows that α is uniformly bounded. Assume that the α-graph is connected. By Lemma 3.7, N is an upper bound on the order of α. This upper bound is tight, i.e. N is indeed the order of α.

Assume to the contrary that for some s, t, with $s<t<N$, $\alpha_t \leq_r \alpha_s$. Lemma 3.1 implies that there exists a homomorphism $h : \alpha_s \rightarrow \alpha_t$. By Lemma 3.8, the α_s-graph is of the form of Figure 3.8. The variables that h maps to themselves are the ones in H'.

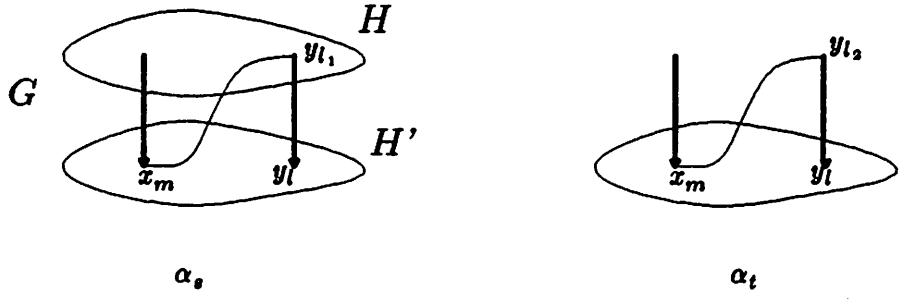

Figure 3.8. Expansions α_s and α_t with $\alpha_t \leq_r \alpha_s$ and maximum path-weight in the corresponding graphs greater than 1.

Since $s<N-1$, Lemma 3.10 guarantees the existence of some path of weight greater than 1, in the α_s-graph. Hence, there exists a path of weight zero with one end in H and the other, a tail of a dynamic edge, in H'. In Figure 3.8, x_m to y_{l_1} is such a path. Because of h, there exists a path of weight zero in the α_t-graph also, like the one from x_m to y_{l_2} in Figure 3.8. Substitution (3.5) implies that for l_1 and l_2

$$l_1 = s+l+1 \qquad l_2 = t+l+1. \tag{3.12}$$

By Lemma 3.6, in the α-graph there exist two walks between the dynamic components where x_m and y_l belong (see Figure 3.9).

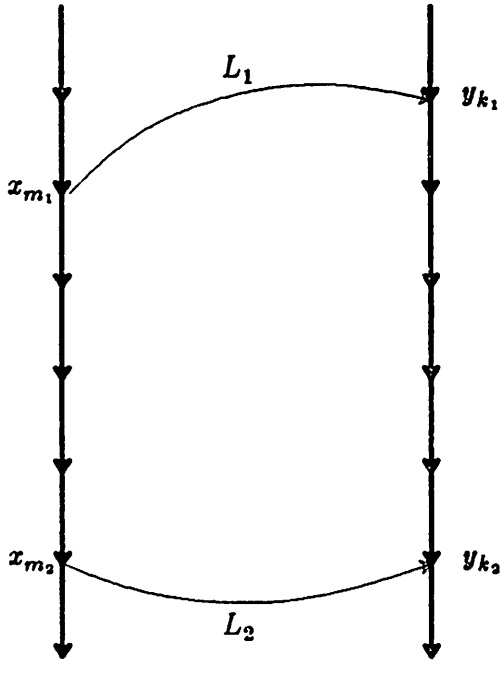

Figure 3.9. Cyclic walk in the α-graph.

Let L_1 and L_2 be the weight of these walks respectively. Figure 3.9 shows that a cyclic walk of weight

$$L = (m_1-m_2)+L_2-(k_1-k_2)-L_1 \tag{3.13}$$

exists in the α-graph. Furthermore, Lemma 3.6 implies that the following hold:

$$(m-m_1) = (l_1-k_1)+(s+1)0-L_1$$

$$(m-m_2) = (l_2-k_2)+(t+1)0-L_2.$$

Subtracting the two above we get $(m_1-m_2)+L_2-(k_1-k_2)-L_1 = (l_2-l_1)$. Making the substitution in (3.13) and using (3.12) yields $L = (t-s)$. Since $t > s$, the cyclic walk in the α-graph has nonzero weight. This implies that there exists some cycle in the α-graph with nonzero weight also, which contradicts the hypothesis. Thus, $\alpha_t \leq_r \alpha_s$ holds for no s, t, with $s < t < N$. The smallest such numbers are $s = N-1$ and $t = N$. Hence, the order of α is N, the index is $N-1$, and the period is 1.

If the α-graph is not connected the above is true for each one of its components. Applying Lemma 3.4, gives again that the maximum path-weight in the complete α-graph is the order of α. □

Example 3.5: The proofs of Theorems 3.2 and 3.3 are illustrated with the following example. Consider the simple recursive Horn clause α:

α : $\mathbf{P}(u_1,u_2,u_4,u_4,y) \wedge \mathbf{Q}(u_1,u_2) \wedge \mathbf{R}(u_2,u_3,x) \wedge \mathbf{S}(w,z) \wedge \mathbf{T}(v) \rightarrow \mathbf{P}(v,w,x,y,z)$.

The α-graph appears in Figure 3.10.

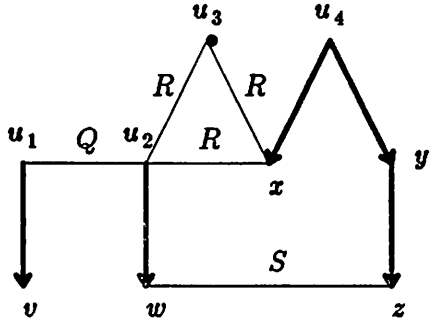

Figure 3.10. The α-graph.

All the cycles in the α-graph have zero weight. Hence, according to Theorem 3.2, α is uniformly bounded. The maximum path-weight in the graph being 2, it implies that α_2 is redundant, i.e. α is equivalent to α_0 and α_1. This becomes apparent by looking at α_1 and α_2.

$$\alpha_1 : \mathbf{P}(u'_1,u'_2,u'_4,u'_4,u_4) \wedge \mathbf{Q}(u'_1,u'_2) \wedge \mathbf{R}(u'_2,u'_3,u_4) \wedge \mathbf{S}(u_2,y) \wedge \mathbf{T}(u_1)$$
$$\wedge \mathbf{Q}(u_1,u_2) \wedge \mathbf{R}(u_2,u_3,x) \wedge \mathbf{S}(w,z) \wedge \mathbf{T}(v) \rightarrow \mathbf{P}(v,w,x,y,z)$$

$$\alpha_2 : \mathbf{P}(u''_1,u''_2,u''_4,u''_4,u'_4) \wedge \mathbf{Q}(u''_1,u''_2) \wedge \mathbf{R}(u''_2,u''_3,u'_4) \wedge \mathbf{S}(u'_2,u_4) \wedge \mathbf{T}(u'_1)$$
$$\wedge \mathbf{Q}(u'_1,u'_2) \wedge \mathbf{R}(u'_2,u'_3,u_4) \wedge \mathbf{S}(u_2,y) \wedge \mathbf{T}(u_1)$$
$$\wedge \mathbf{Q}(u_1,u_2) \wedge \mathbf{R}(u_2,u_3,x) \wedge \mathbf{S}(w,z) \wedge \mathbf{T}(v) \rightarrow \mathbf{P}(v,w,x,y,z).$$

The corresponding graphs are shown in Figures 3.11 and 3.12 respectively. The α_1-graph in Figure 3.11 has two (connected) components, none of which can be the image of the α-graph under any homomorphism. This implies that there are some instances of the relations in α that make α_1 produce some tuples that are not produced by α_0. Hence, α_1 is necessary. On the other hand the α_2-graph in Figure 3.12 has three components. Two of them are homomorphic images of those in the α_1-graph. Therefore, α_2 is not necessary.

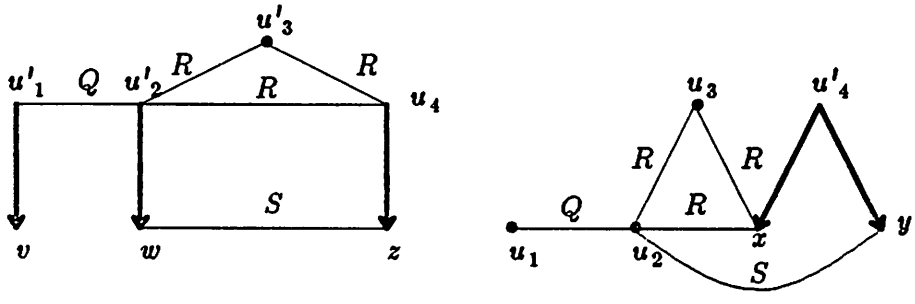

Figure 3.11. The α_1-graph.

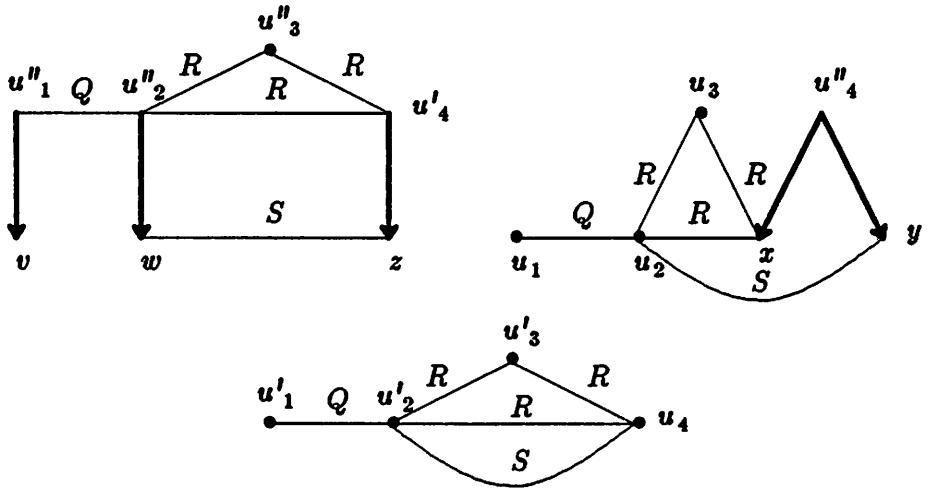

Figure 3.12. The α_2-graph.

□

3.5.3. Necessity of the Condition

The converse of Theorem 3.2 is proved using the following lemmas.

Lemma 3.11: Let α be a recursive Horn clause. If α is uniformly bounded then, for all $k \geq 0$, α_k is uniformly bounded as well.

Proof: In what follows the fact that $(\alpha_k)_l = (\alpha_l)_k = \alpha_{(k+1)(l+1)-1} = \alpha_{k+l(k+1)}$ is used.

Lemma 3.3 implies that $\alpha_N \leq_r \alpha_n$, for some $0 \leq n \leq N-1$. Consider α_k for some $k \geq 0$. Then,

$$
\begin{aligned}
(\alpha_k)_N &= \alpha_{N+k(N+1)} \leq_r \alpha_{n+k(N+1)} && \text{Lemma 3.2} \\
&= \alpha_{n+k(n+1)+k(N-n)} \\
&\leq_r \alpha_{n+k(n+1)} && \text{Lemma 3.2 and transitivity of } \leq_r \\
&= (\alpha_k)_n.
\end{aligned}
$$

Lemma 3.3 implies that α_k is uniformly bounded. □

Consider a typical cycle in the α-graph (see Figure 3.13). It is essentially the path of Figure 3.7, with one more static edge labeled Q_m between x_{m,l_m} and x_{1,k_1}.

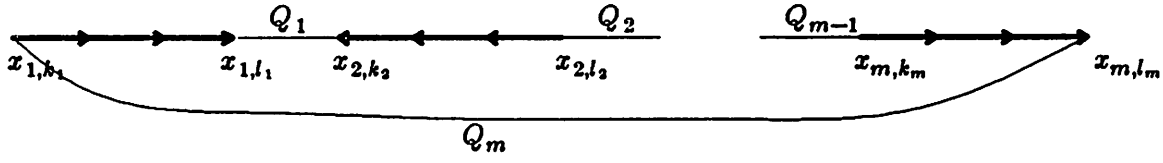

Figure 3.13. Typical cycle in the α-graph.

The following two lemmas are similar to Lemma 3.10.

Lemma 3.12: Let α be a simple recursive Horn clause. If the α-graph contains a cycle of weight $n+1$, $n \geq 0$, with ϵ static edges, then the α_n-graph contains a cycle of weight 1 with ϵ static edges also.

Proof: Adding the static edge labeled $\mathbf{Q}_m$ to the path of Figure 3.7 creates the cycle in Figure 3.13, of weight equal to that of the original path with $\epsilon{=}m$ static edges. Hence, Lemma 3.10 is directly applicable. Since the α-graph contains a cycle of weight $n+1$, it implies that the α_n-graph contains a cycle of weight $\lfloor \frac{n+(n+1)}{n+1} \rfloor = \lfloor \frac{2n+1}{n+1} \rfloor = 1$. The number of the static edges in the cycle remains the same, that is $\epsilon{=}m$. □

Lemma 3.13: Let α be a simple recursive Horn clause. If the α-graph contains a cycle of weight 1 with ϵ static edges, then the α_n-graph, $n \geq 0$, contains a cycle of weight 1 with $\epsilon(n+1)$ static edges.

Proof: The proof is similar to that of Lemma 3.10, so it is only sketched here. The $\mathbf{Q}_i$'s are partitioned into $n+1$ partitions, each partition having exactly one

instance of each Q_i, $1 \leq i \leq m$. The r-th partition, $0 \leq r \leq n$, contains $Q_i(x_{i,r_i+l_i}, x_{i+1,r_i+k_{i+1}})$, such that $r_i = f_{n+1}(r+\sigma_i)$. Each one of these partitions is shown to form a path. Furthermore, the last node of partition r and the first node of partition $r+1$ are connected by a dynamic path, and so are the last node of partition n and the first node of partition 0. Hence a cycle is formed. Its weight is calculated similarly as in Lemma 3.10 and is equal to 1. The static edges in the cycle are those of all the partitions. Each partition has $\epsilon = m$ edges, and there are $n+1$ partitions. Hence, the cycle formed has $\epsilon(n+1)$ static edges. □

Theorem 3.4: Let α be a simple recursive Horn clause. If α is uniformly bounded then the α-graph contains no cycle of nonzero weight.

Proof: Suppose to the contrary that the α-graph contains some nonzero weight cycles. Consider the one with the smallest number of static edges, say ϵ. Let $n+1$, $n \geq 0$, be its weight. By Lemma 3.12, the α_n-graph contains a cycle of weight 1, with ϵ static edges also. Lemma 3.11 implies that α_n is uniformly bounded. Hence, there are two expansions $(\alpha_n)_s$ and $(\alpha_n)_t$, $s<t$, such that $(\alpha_n)_t \leq_r (\alpha_n)_s$. By Lemma 3.1, there exists a homomorphism $h : (\alpha_n)_s \rightarrow (\alpha_n)_t$. Lemma 3.13 implies that the $(\alpha_n)_s$-graph contains a cycle of weight 1 with $(s+1)\epsilon$ static edges, which h maps to another cycle of weight 1 in the $(\alpha_n)_t$-graph (see Figure 3.14).

The cycle contains at most as many static edges as the cycle of the $(\alpha_n)_s$-graph. It may contain fewer static edges if h is not one-to-one but it can never contain more. By corollary 1, there exists a cyclic walk in the α-graph of weight $(n+1)(t+1)-1+1 = (n+1)(t+1)$. Without loss of generality, assume that it is formed

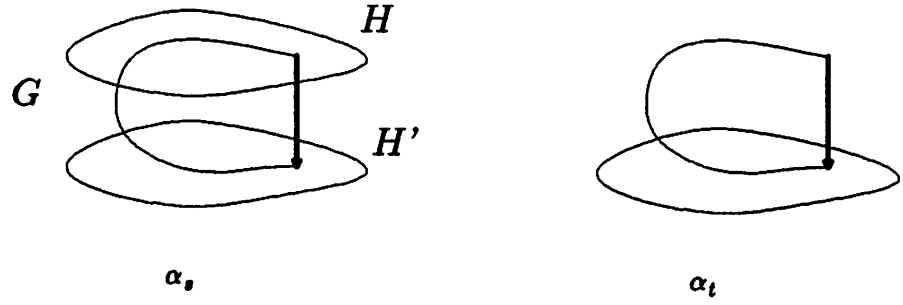

Figure 3.14. Expansions α_s and α_t with $\alpha_t \leq_r \alpha_s$ and cycles in the corresponding graphs of weight 1.

by traversing c times a cycle of weight n' with ϵ' static edges. (the case that it is formed by traversing multiple cycles connected with each other is a trivial extension of what follows). Hence,

$$c\,n' = (n+1)(t+1). \tag{3.14}$$

There are $c\,\epsilon'$ static edges in the cyclic walk, and as mentioned above

$$c\,\epsilon' \leq (s+1)\epsilon. \tag{3.15}$$

Combining (3.14) and (3.15) yields

$$\frac{\epsilon}{\epsilon'} \geq \frac{(n+1)}{n'}\frac{(t+1)}{(s+1)}. \tag{3.16}$$

However, from Lemma 3.2 and the transitivity of $\leq_r$, t may be chosen arbitrarily large. In (3.16) s and n are fixed, whereas there is an upper bound on the value of n', imposed by the form of the α-graph. Hence, there exists some t, satisfying the desired properties, such that $\frac{(n+1)}{n'}\frac{(t+1)}{(s+1)} > 1$. This combined with (3.16) yields $\epsilon > \epsilon'$, that is there exists a cycle in the α-graph, with fewer static edges than ϵ. This contradicts the hypothesis. Hence, all the cycles in the α-graph are of weight zero. □

Example 3.6: Theorem 3.4 is illustrated with an example. Consider the Horn clause below:

$$\beta: \quad \mathbf{P}(u_1,w,u_2,u_3) \wedge \mathbf{Q}(w,u_2) \wedge \mathbf{R}(y,u_3) \wedge \mathbf{S}(x,z) \rightarrow \mathbf{P}(w,x,y,z).$$

The β-graph appears in Figure 3.15.

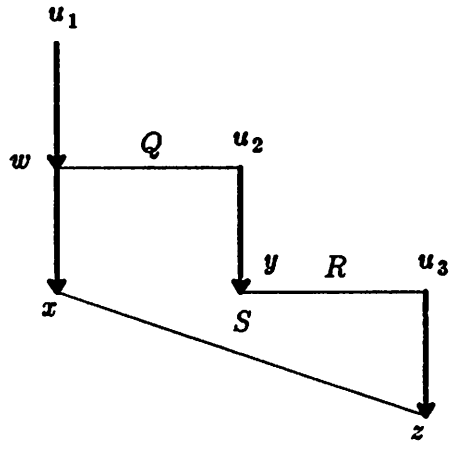

Figure 3.15. The β-graph.

The β-graph contains a cycle of weight 1, namely $(w \rightarrow u_2 \rightarrow y \rightarrow u_3 \rightarrow z \rightarrow x \rightarrow w)$. According to Theorem 3.4, β is not uniformly bounded. This becomes apparent by looking at the graphs of the expansions of β. The β_1- and β_2-graphs appear in Figures 3.16 and 3.17 respectively. Contrary to what happened to the graphs of uniformly

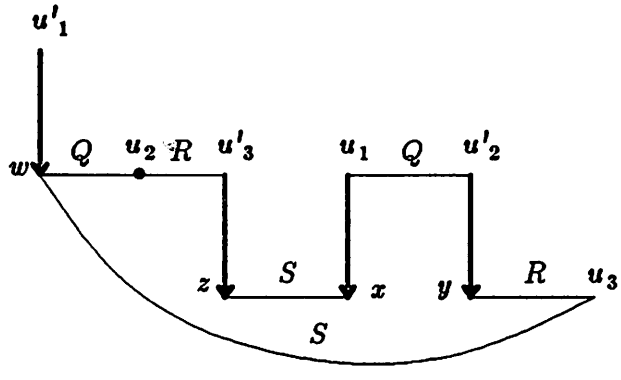

Figure 3.16. The β_1-graph.

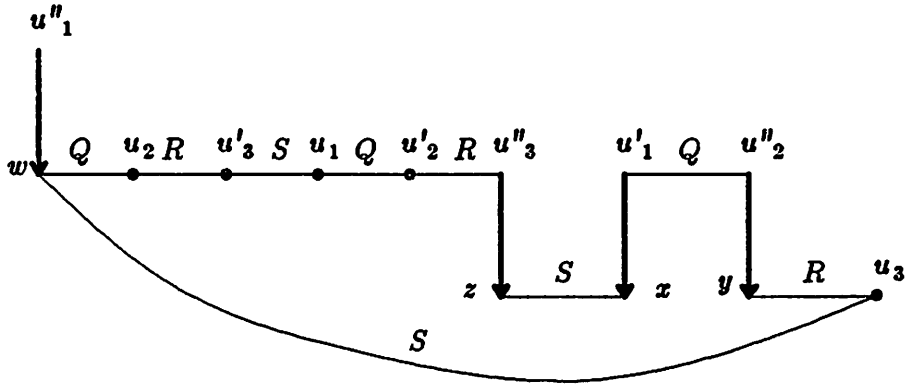

Figure 3.17. The β_2-graph.

bounded Horn clauses, the graphs of the expansions of β continue to have a single component, but the number of static edges in the original cycle increases (Lemma 3.13). This continues, no matter how many expansions are taken. □

Theorem 3.1: A simple recursive Horn clause α is uniformly bounded iff the α-graph contains no cycle of nonzero weight. In that case the order of α is equal to the maximum path-weight in the α-graph.

Proof: The proof follows immediately from Theorems 3.2, 3.3, and 3.4. □

The condition of Theorem 3.1 is sufficient for a Horn clause to be uniformly bounded, even when restrictions R3 to R5 of Section 3.2 are removed. It is not necessary, however, as the following example shows.

Example 3.7: Consider the following Horn clause:

$$\mathbf{P}(y,x) \rightarrow \mathbf{P}(x,y).$$

Clearly, this Horn clause is not simple. It violates restriction R5, since (x,y) is a permutation of (y,x). The graph of the Horn clause appears in Figure 3.18.

Figure 3.18. Graph violating restriction R5.

As expected, the dynamic graph (restricted to the positive edges) is not a forest. Even though it is clear that the Horn clause is bounded with bound 1, the graph contains a cycle of weight 2, thus violating Theorem 3.1. □

3.6. Algorithms

An efficient algorithm exists to test the condition of Theorem 3.1 on the graph of some simple recursive Horn clause. It is a depth-first search algorithm on the complete graph, i.e. with all positive and negative dynamic edges. The number zero is assigned to the first node visited in each component. Every time some edge is traversed, its head is assigned the number assigned to its tail, increased by the weight of the edge. If at any point a node is assigned two different numbers, then there is a nonzero weight cycle in the graph, and therefore the Horn clause is not uniformly bounded. Otherwise it is. The maximum path-weight within a component is equal to the maximum number minus the minimum number assigned to any of its nodes. The maximum such number over all the components is the order of the Horn clause. The algorithm just described appears below.

Input: Graph $G=(V,E,W)$ corresponding to some simple recursive Horn clause α, where V is the set of nodes, E the set of edges, and W the mapping of every edge to its weight; the graph is represented by adjacency lists $L[v]$, for $v \in V$.

Output: If α is uniformly bounded then give its order, otherwise give "UNBOUNDED".

Algorithm: In the following, $D[v]$ denotes the number assigned to v, as described above, and $W[v,w]$ denotes the weight of the edge $(v \rightarrow w)$.

```
begin
    bound:=0;
    for all v in V do mark v "new";
    while there exists a vertex v in V marked "new" do
        begin
            maxpos:=0; maxneg:=0;
            SEARCH (v,0);
            bound:=MAX(bound,maxpos+maxneg)
        end
    output(bound);
end
procedure SEARCH (v,count):
    begin
        mark v "old";
        D[v] := count;
        maxpos := MAX(maxpos,count);
        maxneg := MAX(maxneg,-count);
        for every vertex w in L[v] do
            if w is marked "new" then SEARCH (w,count+W[v,w])
            elseif D[w] ≠ count+W[v,w] then output ("UNBOUNDED");
            exit;
    end
```

Algorithm 3.1. Decision procedure for uniform boundedness.

Lemma 3.14: Algorithm 3.1 returns "UNBOUNDED" iff its input graph has some cycle of nonzero weight; otherwise, it returns the maximum path-weight in the graph. Its time complexity is $O(\nu+\epsilon)$, where ν is the number of nodes and ϵ is the number of edges in the graph.

Proof: The previous discussion proves the correctness of the algorithm. The algorithm is a depth-first search on a graph, with a constant number of operations in

each step. Therefore, its running time is $O(\nu+\epsilon)$ [Aho74]. □

3.7. Transitive Closure

Unfortunately, some useful recursive Horn clauses are not simple. A characteristic example is the transitive closure **P** of a binary relation **Q** expressed by the Horn clause

$$\mathbf{P}(x,z) \wedge \mathbf{Q}(z,y) \rightarrow \mathbf{P}(x,y).$$

This is clearly unbounded and one would expect to be able to characterize uniform boundedness for Horn clauses of this form. To achieve that we relax restriction R5. We define a *permutation* Horn clause to be one whose corresponding graph *is* a dynamic cycle. The dynamic α-graph of a simple recursive Horn clause is a forest. Restriction R5 is relaxed by allowing components of the dynamic graph to *be* cycles, as long as there are no static edges attached to them in the complete graph. That is, each component of the α-graph is either simple or permutation.

Example 3.8: Let α be the above given Horn clause representing the transitive closure of Q:

$$\alpha: \qquad \mathbf{P}(x,z) \wedge \mathbf{Q}(z,y) \rightarrow \mathbf{P}(x,y).$$

The α-graph is shown in Figure 3.19.

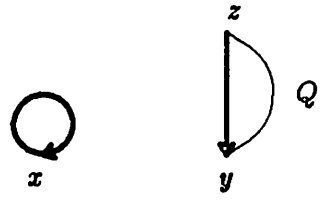

Figure 3.19. The graph corresponding to the transitive closure of a binary relation.

Clearly, α belongs in the new extended class of Horn clauses, since the one component in the α-graph is simple and the other is permutation. □

The following theorem characterizes uniform boundedness for the new extended class of recursive Horn clauses.

Theorem 3.5: Let α be a recursive Horn clause in the extended class. It is uniformly bounded iff the simple components of the α-graph contain no cycle of nonzero weight. In that case, α has order $N = N_s + lcm\{P_i\}$ and period $P = lcm\{P_i\}$, where N_s is the maximum path-weight in the simple components of the α-graph, and $\{P_i\}$ is the set of weights of the cycles in the permutation components.

Proof: Consider a recursive Horn clause α in the extended class. Assume that the α-graph contains M permutation components, with P_i dynamic edges in the i-th one, $1 \leq i \leq M$. Each such component is clearly uniformly bounded, since it simply permutes distinguished variables. Apparently, period is well defined for permutation components, and for the i-th component is P_i. Its order is $P_i - 1$. Notice that the index is equal to $(P_i - 1) - P_i = -1$. This actually represents the identity Horn clause, with antecedent equal to the consequent, and may be denoted as α_{-1} for consistency. Furthermore, by Theorem 3.4, each simple component is uniformly bounded iff it contains no nonzero weight cycles. In that case it has been shown that the order of the component is the maximum path-weight in the graph of the component and the period is 1. Applying Lemma 3.4 yields that α is uniformly bounded iff the simple components of the α-graph contain no nonzero weight cycles. In that case, if the maximum path-weight in the simple components is N_s, then Lemma 3.4 implies that the

period P of α is equal to $P = lcm\{P_i\}$ and the order N is $N = N_s + lcm\{P_i\}$. □

3.8. Applications

Besides its theoretical interest, characterizing boundedness has implications on general recursive Horn clause processing also. For example, assume that a recursive Horn clause can be decomposed into "smaller" ones, some of which are uniformly bounded. The ones that are unbounded are smaller than the initial one and this results in faster processing. Furthermore, the parts of the result that correspond to uniformly bounded Horn clauses are obtained in a fixed number of steps independent of the rest of the Horn clauses. This results in greater efficiency, since processing the original Horn clause may involve more steps than the order of its bounded components. Processing the original Horn clause recomputes the same things again and again. Of course, there is some overhead in the end to combine the results of the various Horn clauses so that the same result as the original Horn clause is produced. In many cases, however, there is the potential for some net savings in computational cost.

Example 3.9: As an example of such a decomposition consider the following Horn clause:

$$\mathbf{P}(z,w) \wedge \mathbf{Q}(z,w) \wedge \mathbf{R}(x,y) \wedge \mathbf{S}(z,x) \rightarrow \mathbf{P}(x,y).$$

The graph of this Horn clause is shown in Figure 3.20. The Horn clause can be decomposed into the two Horn clauses

$$\mathbf{P}_1(z,w) \wedge \mathbf{Q}(z,w) \wedge \mathbf{R}(x,y) \rightarrow \mathbf{P}_1(x,y)$$

$$\mathbf{P}_2(z) \wedge \mathbf{S}(z,x) \rightarrow \mathbf{P}_2(x).$$

The corresponding graphs of the two Horn clauses appear in Figure 3.21.

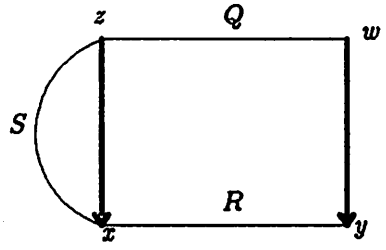

Figure 3.20. Graph of decomposable Horn clause.

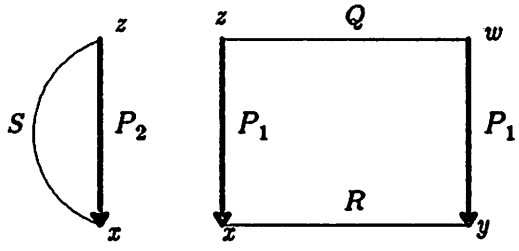

Figure 3.21. Graphs of Horn clauses after decomposition.

The first Horn clause is uniformly bounded with order 1, whereas the second one is unbounded. Processing the two Horn clauses separately and then combining the two results may affect the processing time significantly. □

With respect to processing time, uniformly bounded Horn clauses have several advantages. They can be expressed nonrecursively in a finite form. Hence, all the tools used in conventional relational databases to find fast access paths are applicable. Compiling such an access path is much easier compared to the effort needed for an unbounded Horn clause [Hens84]. Finally, if a uniformly bounded Horn clause with order N is processed by some iterative program, only N iterations through the loop are needed. Iterating for an $(N+1)$-th time produces no new tuples. For Horn clauses with small orders, say 1 or 2, the savings may be significant.

3.9. Summary

Recursion is a major source of inefficiencies in a deductive database system. In this sense, identifying cases where recursion may be removed becomes important. For a restricted class of linear recursive Horn clauses, necessary and sufficient conditions are given for a member of the class to be equivalent to a finite number of nonrecursive Horn clauses, i.e. to be uniformly bounded. This has been accomplished by modeling the Horn clause with a weighted directed graph. Removal of recursion is possible if and only if there is no nonzero weight cycle in the graph. The existence of an efficient (linear time) algorithm for the decision procedure makes the result easy to apply. Finally, when decomposition is a possibility, identifying parts of a Horn clause that are uniformly bounded may gracefully affect the processing time.

CHAPTER 4

PROCESSING ALGORITHMS

In the previous chapter, the class of linear recursive Horn clauses that are equivalent to a finite set of nonrecursive ones was examined. It was mentioned that in this case recursion can be removed and all queries on the recursive relation can be answered by a regular relational query optimization and processing engine. However, the vast majority of linear recursive Horn clauses do not have this property. The mere fact that only the number 0 as a weight for all cycles in the graph corresponding to the Horn clause guarantees its uniform boundedness, whereas all the other integers do not, is already a strong indication of the limited scope of the applicability of the boundedness criterion to process recursion. So, we now turn into addressing the general problem of query processing in the presence of recursively defined relations.

4.1. Previous Work

Recently, there has been an extensive effort to develop techniques for answering queries on recursively defined relations. In [Banc86b] one may find a good survey of an almost complete subset of them. Detailed descriptions of the algorithms can be found in [Aho79a, Hens84, Ullm85, Lozi85, Banc85, Banc86a, Han86, Viei86, Demo86]. Also, some of the first attempts to evaluate queries on recursively defined relations can be found in [Gall78, Gall81b, Gall83]. We give a synoptic overview of the known approaches, while still avoid getting into the details of the algorithms that are

unnecessary for our presentation.

Although most of the known approaches are based on the Horn clause formulation of recursion, here they are described using the algebraic framework of Chapter 2. Also, only linear and immediate recursion is examined, even though some of the techniques are applicable to the more general case as well. With these assumptions it has been shown in Chapter 2 that a recursively defined relation can be seen as the minimal solution of an equation of the form

$$A\mathbf{P} \cup \mathbf{Q} = \mathbf{P}, \tag{4.1}$$

with $A \in R$ a multiplication of primitive relational operators and $\mathbf{Q}$ a stored relation in the database. The minimal solution of (4.1) is of the form $\mathbf{P} = A^*\mathbf{Q}$. Therefore, any query on relation $\mathbf{P}$ can be transformed to a query on $A^*\mathbf{Q}$. For example, the query $\mathbf{P}(c,...)?$ that asks for the tuples of $\mathbf{P}$ that have the constant c in the first column p_1 will be the query $\sigma_{p_1=c}A^*\mathbf{Q}$, which is a query on base relations. In the following descriptions of the algorithms the stored relation $\mathbf{Q}$ is ignored. Their description is in terms of computing A^*. Applying that on $\mathbf{Q}$ is straightforward.

There are three major issues that arise in processing queries on recursively defined relations, that any appropriate algorithm must address and solve.

- The first issue is determining the minimal number of terms from A^* that are needed to answer the query. The importance of solving it is twofold. First, the algorithms actually stop after a finite number of steps. Otherwise, depending on the given query, they may run forever without producing the answer, somehow trying to create all infinite number of terms of A^* (an example of such an algo-

rithm is the one used in Prolog (Section 4.1.6)). Second, as few terms of A^* as possible are generated, avoiding unnecessary computation. It has already been mentioned in Chapter 2 that A^* is equal to only a finite number of its terms for any given database. However, the number of terms used there is, in general, much larger than the one usually needed for any specific instance of the database and any given query. Therefore, when efficiency becomes an issue, knowing when the answer to a given query has been computed is crucial.

- The second issue is avoiding repetition of computations. For example, A^2 may be generated at some point and then, it is generated again to be multiplied with A for the generation of A^3. Clearly, generating A^2 twice in the computation is unnecessary. Identifying and avoiding this kind of inefficiencies is of utmost important for the overall performance of an algorithm.
- The final issue is using the possible selections of the query to guide the execution. In the presence of a selection, only a subset of the underlying database is needed to produce the answer. Using selections to minimize the amount of data at which the system has to look to answer a query has proved crucial in ordinary relational query processing [Wong76, Seli79], and it is even more so in the recursive case.

In the following subsections an abstract view of the algorithms proposed in the literature is given. The algorithms described are referred to as Naive and Semi-Naive [Banc85], Shapiro-McKay [Shap80], Query-Subquery (QSQ) [Viei86], Prolog [Cloc81], Aho-Ullman [Aho79a], Kifer-Lozinskii [Lozi85], Henschen-Naqvi [Hens84], Han-Lu

[Han86], Demo [Demo86], Magic Sets [Banc86a], and Counting [Banc86a].

Each algorithm is based on a particular way of writing A^* within the closed semiring E_R of the relational operators. In that sense, the algorithms are classified into the following three categories, according to the type of algebraic transformations applied on A^* for the algorithm to be realized:

1) *Syntactic:* These are transformations that are based solely on the properties of the closed semiring E_R. The operators appearing in A^*, their parameter relations and the constants in the query are uninterpreted. So, these transformations are always applicable.

2) *Hybrid:* These are transformations that are based on individual properties that the specific operators have. In other words, the operators are interpreted for these transformations to be applicable. However, the relations and the constants in the query are still uninterpreted and any value for them is acceptable.

3) *Semantic:* These are transformations for which the operators, the relations, and the query constants are interpreted. For each instance of the relations or value of the constants a different operator results from the transformation. The transformations include accessing the database and creating relations based on the contents of the database and the constants in the query. These relations are then used as parameters to other operators, which behave differently for different instances of their parameters. It is in this spirit that we call these transformations semantic. Otherwise, all the operators can be constructed by purely syntactic means.

4.1.1. Syntactic Transformations

• Naive Evaluation

The name of the algorithm is due to Bancilhon [Banc85] and refers to the algorithm proposed in [Aho79a]. It has already been shown in Chapter 2 that $\sum_{k=0}^{n} A^k = (1 + A)^n$. The naive evaluation is a direct application of this form:

$$A^* = \lim_{k \to \infty} (1 + A)^k = \cdots (1 + A)(1 + A) \tag{4.2}$$

(whenever explicit parenthesization is omitted, right associativity is assumed for multiplication). In other words, A is applied on some initial relation $\mathbf{Q}$ and then it is applied on the union of the result with $\mathbf{Q}$, etc., until nothing new gets produced. This is a very simple but highly inefficient algorithm because operator A is applied on the same tuples again and again. This can be seen if $(1 + A)^n$ is treated as if A were a real number. It would then be

$$(1 + A)^n = \sum_{k=0}^{n} \begin{bmatrix} n \\ k \end{bmatrix} A^k,$$

where $\begin{bmatrix} n \\ k \end{bmatrix}$ are the combinations of k items out of n (n choose k). Proposition 2.4a states that $A + A = A$, so the binomial coefficients are eliminated. The naive evaluation actually computes A^k as many times as its binomial coefficient above, that is $\begin{bmatrix} n \\ k \end{bmatrix}$, which is clearly unnecessary†.

† If duplicates were removed after each multiplication then A^k would be computed only $n+1-k$ times. This is better than what is shown above, but it still includes unnecessary computation.

• **Semi-naive Evaluation**

Consider the original form for A^*, that is $A^* = \sum_{k=0}^{\infty} A^k$. Clearly, it is more practical than the one for the naive evaluation in the sense of being faster. Each power of A is computed only once. Exactly this form has been followed in [Gutt84] and elsewhere. Also, this form has been formally extracted using a differential approach in [Baye84] and [Banc85], even for more general cases of recursion, i.e. without any assumptions about linearity. In fact, the name to the algorithm was given in the latter reference. To understand the algorithm, the above equation is written as

$$A^* = \sum_{k=0}^{\infty} A^k = 1 + (1 + (1 + (\ \cdots\ )A)A)A. \tag{4.3}$$

In this form, in each step A is applied only on the tuples produced during the previous step, so unnecessary computation is avoided.

• **Shapiro-McKay**

Another algorithm that is based on similar transformations of A^* has been proposed by Shapiro and McKay [Shap80]. They perform the minimum number of multiplications of operators, at the expense of not taking into account the possible selections in the query to reduce execution time. Their computation scheme applies semi-naive evaluation to compute A^* (using A^k to compute A^{k+1}), and then applies the selection. This delay in applying the selection, results in long execution times for many queries.

• **Query-Subquery (QSQ)**

A method that tries to take into account the constants of the query as much as possible has been proposed by Vieille [Viei86] under the name Query/Subquery. Consider computing σA^*. Assume that

$$\sigma A = \sigma', \quad \sigma A^2 = \sigma' A = \sigma'', \cdots,$$

where the selections σ, σ', σ'' have a more general meaning, i.e. they are selections with multiple values for the selected columns. In this sense, one may think of them as joins with a single column relation containing the selected values. From the above

$$\sigma A^* = \sigma \sum_{k=0}^{\infty} A^k = \sigma + (\sigma A) \sum_{k=0}^{\infty} A^k = \sigma + \sigma' \sum_{k=0}^{\infty} A^k$$

$$= \sigma + \sigma' + (\sigma' A) \sum_{k=0}^{\infty} A^k = \sigma + \sigma' + \sigma'' \sum_{k=0}^{\infty} A^k = \cdots$$

In this form, σA^* is applied on some stored relation Q to generate the answer to the query. For the actual implementation of the algorithm, a recursion control mechanism is used to ensure termination and avoidance of repeated work (e.g. σ' is generated only once and used for the generation of σ'' etc.).

In [Viei86] a second method is described, which uses an iterative scheme to answer the query. It propagates the selection as the previous method. However it uses the form $A^* = \lim_{n \to \infty} (1 + A)^n$ as its underlined processing algorithm (i.e. naive evaluation), instead of $A^* = \sum_{k=0}^{\infty} A^k$ (semi-naive evaluation) that the recursive method does. Hence, it is much less efficient, as it has been pointed out in [Viei86] as well.

• **Prolog**

The programming language Prolog [Cloc81] uses a method almost identical to the recursive QSQ of the previous section for query processing. The main difference is that the processing scheme is tuple-at-a-time instead of a set-at-a-time, and this generates additional inefficiencies, especially when the database is stored on disks. Since its execution paradigm is the same as that of QSQ we discuss it no further.

4.1.2. Hybrid Transformations

• **Aho-Ullman**

The naive evaluation was among the first proposals to incorporate recursion in database systems [Aho79a] (see Section 4.1.1). That proposal included some transformations that occasionally could be applied on queries with selections to get the answer faster. For example, consider a query with a selection σ on the produced relation. The answer to the query will have the form

$$\sigma(1 + A)^N.$$

In [Aho79a] various cases are identified for which σ and A commute, that is $\sigma A = A\sigma$. Whenever this is true, it may be applied inductively on the solution above and get transformed into

$$(1 + A)^N\sigma,$$

which is presumably more efficient since the operators are applied on smaller relations.

Equivalently, start from the initial equation $B = 1 + AB$. Having a selection σ in the query is equivalent to asking for an operator σB. Thus,

$$\sigma B = \sigma(1 + AB) = \sigma + \sigma AB = \sigma + A(\sigma B).$$

This equation has almost the same form as the original one, having σB as the unknown and σ as the constant operator (instead of 1). In Example 2.2, it has been shown that the solution to the above equation is $A^*\sigma$, which is equal to $(1 + A)^N\sigma$ for some N when naive evaluation is used.

Pushing the selection through A^*, that is writing σA^* as $A^*\sigma$, is not always possible. We have used $\sigma A = A\sigma$ to make it. In [Deva86] some necessary and sufficient conditions for a generalized version of the problem are given.

• **Kifer-Lozinskii**

In [Kife85] an extension of the Aho-Ullman algorithm for using selections to answer the query is presented. They use a graph to have the selections propagate into the query as much as possible. Their improvement mainly consists on having the ability to do this in some cases where the Aho-Ullman approach fails. For example, consider A that can be written as a sum, $A = B + C$. If $\sigma B = B\sigma$ and $\sigma C = C\sigma$ then

$$\sigma(B + C) = \sigma B + \sigma C = B\sigma + C\sigma = (B + C)\sigma.$$

Once again, this implies that $\sigma A^* = A^*\sigma$. The second form is more efficient than the first one, since the operators are applied on smaller relations.

• **Henschen-Naqvi**

A significant contribution to the understanding of query processing with recursively defined relations has been made by Henschen and Naqvi [Hens84]. They propose a considerably general technique to compile programs answering specific queries (involving selections) given a set of Horn clauses. Their goal is to make use of the

selections in the query as early as possible. The main idea of their processing paradigm is described for the following special case. Assume that A is of the form $A = BC$ with B and C such that they commute, i.e. $BC = CB$. Then,

$$A^* = \sum_{k=0}^{\infty} A^k = \sum_{k=0}^{\infty} (BC)^k = \sum_{k=0}^{\infty} B^k C^k.$$

The last equality can be easily proved by induction on k. Given a query with a selection σ on attributes of the relations in B only (neither on relations in C nor on the input relation), it is answered by $\sum_{k=0}^{\infty} (\sigma B^k) C^k$. Repeated computations are avoided by using old results efficiently. That is, σB^k is kept and used for the computation of σB^{k+1}, which is then multiplied $k+1$ times with C to get A^*.

• **Han-Lu**

In [Han86] three algorithms are presented and their performance is analyzed for a particular operator A and a particular query. The algorithms are called "single wavefront", which is actually the Henschen-Naqvi algorithm, "central wavefront", which is the Shapiro-McKay algorithm, and "double wavefront" which is the one we call Han-Lu algorithm. Since the first two have been described above, only the third one is described here. It applies to operators with the properties mentioned in the description of Henschen-Naqvi, that is $A = BC$ with $BC = CB$. Also the selection in the query is on attributes of relations in B only. Again, in this case

$$\sigma A^* = \sum_{k=0}^{\infty} (\sigma B^k)(C^k).$$

The difference of their algorithm from Henschen-Naqvi is that not only σB^k is kept to be used for the computation of the σB^{k+1}, but also C^k is computed alone and used in

the next step to compute C^{k+1}. This algorithm has potential inefficiencies since C^k is computed without the graceful effect of a selection as in Henschen-Naqvi. However, analytical as well as experimental studies have shown that there are many cases for which that it performs better than Henschen-Naqvi [Han86].

• **Demo**

In [Demo86] one iterative and one recursive method for processing queries with selections on recursively defined relations are described. It is pointed out that depending on the selection, one of the methods is preferred over the other. The iterative method is very close to Aho-Ullman and the recursive method resembles Henschen-Naqvi so they are described no further.

4.1.3. Semantic Transformations

• **Magic Sets**

Magic sets were introduced in [Banc86a]. Even though the notion applies to more general cases as well, we assume that the transformation of the magic sets does not introduce any mutual recursion (for a more general example see [Banc86b]). The idea behind magic sets is the following: Consider a query $\sigma A^* \mathbf{Q}$. First, a relation **S** is created, the magic set, as $\mathbf{S} = B^* \mathbf{R}$, where B is a new operator that is constructed from the specific selection σ and operator A, and **R** is the relation containing the values in the selection. The answer to the original query is found by computing $\sigma C^*[\mathbf{S}]\mathbf{S}$, where C is yet another operator constructed from A that has the magic set **S** as a parameter (as well as input). By the above transformation, a large portion of the database that contributes nothing to the answer is not accessed. Even though

magic sets are not a panacea [Banc86a], computing $B^*\mathbf{R}$ and then $\sigma C^*[\mathbf{S}]\mathbf{S}$ is often more efficient than applying some of the other algorithms.

• **Counting**

The last method described, which is the second based on semantic transformations of A^*, is called counting and was proposed in [Banc86a] as well. It is an offspring of the magic sets method and transforms the original operators by introducing arithmetic. A counting magic relation $\mathbf{S}$ is created again, as $\mathbf{S} = B^*\mathbf{R}$. The form of the new operator B depends on A and the selection σ, and $\mathbf{R}$ is the relation of the selection values paired with the integer 0. The values generated by $B^k\mathbf{R}$ are paired with k. The answer to the original query is found by computing $\sigma' C^*[\mathbf{S}]\mathbf{S}$, where C is a new operator having $\mathbf{S}$ as a parameter and σ' is σ enhanced with a selection value for the integer column introduced for the counting.

Counting cannot be applied to all the cases that magic sets can. However, whenever applicable it usually performs at least as well as any other method described until now. This has been verified in both [Banc86a] and [Banc86b].

Notice that the above classification into the three categories is very similar to the one in [Banc86b]. There, the algorithms are classified as *methods* (evaluation algorithms), which correspond to the syntactic transformations of our classification, and *optimization strategies*, which correspond to the hybrid and semantic transformations of our classification. The only point were this correspondence fails to hold is in the classification of Henschen-Naqvi, which was a method in [Banc86b] and a hybrid

transformation here. The discrepancy is only due to the different viewpoint of the two classifications.

4.2. New Algorithms

By looking at all the algorithms described in Section 4.1, one clearly gets the idea that, apart from the various transformations that are applied to make use of selections as much and as early as possible, they all use naive or semi-naive evaluation at some abstract level. However, these are not the only ways to perform the task in the sense that $\sum_{k=0}^{\infty} A^k$ and $\lim_{k\to\infty}(1+A)^k$ are not the only ways to express A^*. Therefore, the existence of other evaluation methods of A^* that are more efficient than the traditional ones (naive and semi-naive) is to a large extent an open problem. In this section some new algorithms are presented by transforming A^* in new ways. According to the classification of Section 4.1, these new transformations are classified as syntactic. Analytical and experimental results are also given that indicate that the "traditional" algorithms are not always optimal. In fact, as it becomes apparent in the sequel, using the new algorithms proposed, significant improvement in both I/O and CPU time is observed in many cases.

4.2.1. Equivalent Forms of the Solution

Two equivalent forms for A^* have already been mentioned, namely $\lim_{k\to\infty}(1+A)^k$ and $\sum_{k=0}^{\infty} A^k$, which correspond to the naive and semi-naive evaluation respectively. Some specific examples help in understanding the flow of the algorithms.

Example 4.1: Consider the naive evaluation. Let A be the operator corresponding to the recursive Horn clause defining the **ancestor** relation (Example 1.3 in Section 1.2). Assume that the relation **father** is given by the graph of Figure 4.1, where an edge $(a \rightarrow b)$ indicates a tuple **father**(a,b), i.e. a is the father of b. Consider the query **ancestor**(Uranus,y), asking for Uranus's descendants. Following the steps of the algorithm corresponding to (4.2) the answer is developed as shown in Figure 4.1.

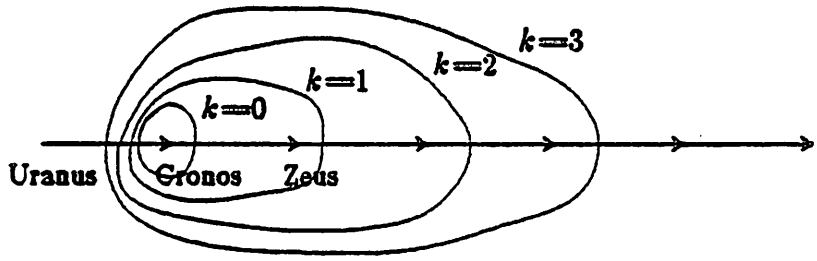

Figure 4.1. Naive algorithm to answer **ancestor**(Uranus,y).

The answer is developed by finding in the k-th step (that is $(1 + A)^k$) Uranus's descendants that are l generations below him, $1 \leq l \leq k+1$, for $k \geq 0$. Initially Uranus's children are generated, which taken together with Uranus are used to find their children, which are Uranus's children and grandchildren together etc. This is clearly not a good processing strategy, since A is applied on the same tuples many times producing the same result again and again. For example, Cronos is produced as a descendant of Uranus in every iteration. □

With respect to efficiency, the form $A^* = \sum_{k=0}^{\infty} A^k$, which corresponds to the semi-naive evaluation, is faster and therefore more practical. Formula (4.3) in Section 4.1.1 gives the form of A^* corresponding to the semi-naive evaluation:

$$A^* = \sum_{k=0}^{\infty} A^k = 1 + (1 + (1 + (\ \cdots\ )A)A)A. \tag{4.3}$$

In this form A is applied each time only on the tuples produced during the previous iteration, so unnecessary computation is avoided.

Example 4.2: For the same example as before, using A^* from (4.3), Uranus's descendants are generated as shown in Figure 4.2.

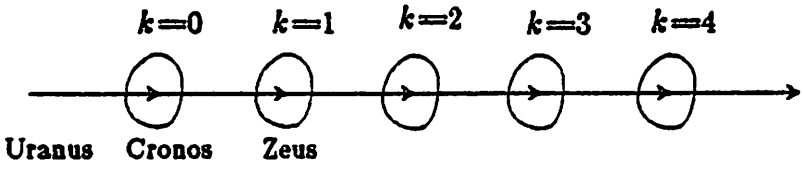

Figure 4.2. Semi-naive algorithm to answer **ancestor**(Uranus,y).

In the k-th step, A^k, $k \geq 0$, Uranus's descendants that are exactly $k+1$ generations below him are found. Initially, Uranus's children are generated, in step 1 their children are generated, which are Uranus's grandchildren, etc. In the end, the union of all the sets is taken to produce the complete answer. □

In the search for other equivalent forms for A^* that possibly suggest more efficient execution algorithms, we arrive at the following form:

$$A^* = \prod_{k=0}^{\infty} (1 + A^{2^k}) = \cdots (1 + A^4)(1 + A^2)(1 + A). \tag{4.4}$$

The algorithm indicated by this form for A^* avoids the application of the same operator on the same tuples more than once, so it is faster than the naive evaluation. The interesting question is how it compares with the second form (4.3) of the semi-naive evaluation. In the same spirit of naming the original two algorithms naive and semi-naive [Banc85], we call the algorithm corresponding to (4.4) *smart algorithm*.

Example 4.3: To get a feeling for the smart algorithm, Uranus's descendants are found again. The steps of the generation of the answer are shown in Figure 4.3.

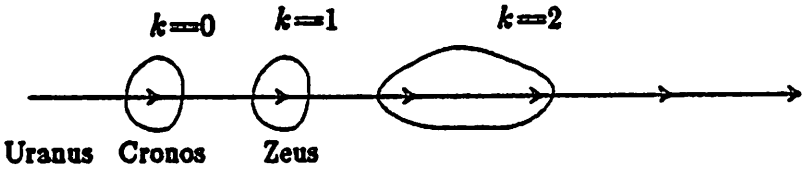

Figure 4.3. Smart algorithm to answer ancestor(Uranus,y).

In the k-th step Uranus's descendants that are l generations below him, $2^{k-1}<l\leq 2^k$, $k\geq 0$ are found. Initially, Uranus's children are generated, which in step 1 are taken them with Uranus to find their grandchildren, which are Uranus's grandchildren and great-grandchildren together, etc. The particular query that was used in the examples is not one where (4.4) is the most efficient as it is pointed out in Section 4.3. It is only presented here because of its simplicity. □

The algorithm corresponding to (4.4) has been independently proposed by Valduriez and Boral [Vald86]. They use a different formalism to describe it, namely Relational Algebra Programs, but it is essentially the same algorithm as (4.4). Also, the smart algorithm is reminiscent of graph theory algorithms to find transitive closure of graphs.

Looking at the three algorithms together, the following observations can be made: At each step, the naive algorithm applies the same operator on all the tuples produced up to that point. The semi-naive algorithm does the same but only on the tuples produced in the last iteration. Finally at each step, the smart algorithm applies a different operator on all the tuples produced up to that point. In this sense, it is the dual of the semi-naive algorithm with respect to the naive algorithm. Figure 4.4

summarizes the above.

	Same Operator	Different Operator
Last iteration result	*SEMI-NAIVE*	
Complete result	*NAIVE*	*SMART*

Figure 4.4. Algorithm types for the computation of A^*.

It is an interesting question to study whether there exists an algorithm covering the last remaining empty box.

The issue raised is whether the smart algorithm runs faster than the others. The formulas alone are not enough to give any useful conclusion in this direction. On the one hand, the number of multiplications performed by the smart algorithm is much smaller than that of the semi-naive one (roughly, it is equal to $2*\log_2 N$, $\log_2 N$ to find the powers of 2 and another $\log_2 N$ for the outer multiplications, with N the number the semi-naive algorithm needs). Under the assumption that in most cases multiplying implies joining, the smart algorithm performs fewer joins. On the other hand, in each step bigger portions of the final outcome are calculated, by applying more expensive operators on larger relations than the semi-naive algorithm. Hence, each step is definitely more expensive. When this trade-off of the number of multiplications versus their individual costs is beneficial for the overall performance, is the focal point of the discussion that follows.

4.2.2. I/O Cost Analysis

In this section, an analysis of the I/O performance of the semi-naive and smart algorithms (forms (4.3) and (4.4)) is presented. For simplicity, the analysis (and the experiments in Section 4.2.3) has been restricted on A representing the computation of the transitive closure of a binary relation. Extensions to more general forms of A are straightforward. For the semi-naive algorithm, it is assumed that all significant powers of A are computed first, as $A^k = AA^{k-1}$, and at the end the sum of all of them is taken. The relation in A is sorted only once in the beginning. Likewise, for the smart algorithm all operators A^{2^k} are computed with the results kept sorted (**loop1**), and then massive joins are performed between the current result and the corresponding power of A, which would be of the form A^{2^k}, for some k (**loop2**). The outcome of these joins is appended directly to the current result to be used in the next iteration.

For the analysis the following parameters are needed.

$tup[k]$	Number of tuples in the relation of A^k
$page[k]$	Number of pages occupied by $tup[k]$ tuples (= $pagenum(tup[k])$, see below)
$pagenum(t)$	Number of pages occupied by t tuples
$pgsz$	Number of tuples fitting in a page
buf	Number of buffers available in the system
$sort(p)$	$= 2p\lceil \log_{buf} p \rceil$, I/O cost to sort p pages having buf buffers [Blas76]
C	I/O cost to create a relation
D	I/O cost to destroy a relation
N	Number of iterations needed by the semi-naive algorithm
M	$= \lfloor \log_2 N \rfloor = \max_k \{k: tup[2^k] \neq 0\}$

Notice that the final outcome for A^* contains $\sum_{k=1}^{N} tup[k]$ tuples. Using the above parameters, the I/O cost *s_naive_io* and *smart_io* of the semi-naive and smart algorithms respectively are calculated as follows:

semi-naive algorithm

s_naive_io =

$sort(page[1])$	Sort original relation on appropriate column(s). It is done only once.
$+ N page[1]$	At each step read sorted original relation.
$+ \sum_{k=1}^{N} sort(page[k])$	Sort the second relation for the join.
$+ \sum_{k=2}^{N} page[k]$	Write the outcome of the join.
$+ \sum_{k=1}^{N} page[k] + pagenum(\sum_{k=1}^{N} tup[k])$	At the end read all the intermediate results and put them into one relation.
$+ N(C + D) + C$	Create and Destroy the N intermediate results and also create the final result.

smart algorithm

smart_io =

	loop1
$2 \sum_{k=0}^{M} sort(page[2^k])$	For each step sort A^{2^k} on two different (sets of) columns for the join. One of them is kept for loop2.
$+ \sum_{k=1}^{M} page[2^k]$	Write the result.
	loop2
$+ \sum_{k=1}^{M} sort(pagenum(\sum_{l=1}^{2^k-1} tup[l]))$	At each step, sort current answer for the next join.
$+ \sum_{k=1}^{M} page[2^k]$	Read second relation (A^{2^k}), which is sorted from loop1.

$+ \sum_{k=1}^{M} pagenum(\sum_{l=1}^{2^k-1} tup[2^k+l])$	Join the two relations and append the outcome to the result.
$+ \sum_{k=1}^{M} page[2^k]$	Append the relation of the used A^{2^k} to the result.
$+ M(C + D) + C$	Create and destroy the M intermediate results and also create the final result.

Before evaluating the above formulas for some specific values of their parameters we make the following comments:

(a) For large relations, the most significant terms for both formulas are the ones of sorting, since the other ones are linear in the size of the input. In the smart algorithm, relations larger than in the semi-naive one are sorted. Hence, it is expected that, as the relations grow, the semi-naive algorithm performs better.

(b) On the other hand, the sorting cost is highly dependent on the number of available buffers. Increasing this number makes the smart algorithm benefit more than the semi-naive one, and therefore makes its performance more competent.

(c) Also, for small relations (where sorting is not that expensive) the overhead of creating and destroying temporaries (costs C and D above) may become significant. In that case, the fact that the factor of $C + D$ in *s_naive_io* is greater than in *smart_io* (N vs. $M = \lfloor \log_2 N \rfloor$) makes the smart algorithm perform better. This is especially true when N is quite large so that there is significant difference with its logarithm.

(d) Finally, most of the time, the smart algorithm overcomputes powers of A that are not significant (they are equal to 0 for the database concerned). This is not

true with the semi-naive algorithm since it computes one power at a time. As N increases the number of joins performed by the smart algorithm changes only when a power of 2 is reached. This makes the algorithm particularly weak at these points, where the overcomputation is maximal.

To validate the observations above we apply the formulas for *s_naive_io* and *smart_io* on some specific cases. The number of the parameters involved is significantly large, so the complete spectrum of possibilities is not covered. Nevertheless, we believe that the special cases examined below are enough to give some insight for the rest also. Recall that A represents the transitive closure of a binary relation. The relations are assumed to be trees. By doing this, we avoid worrying about retaining any possible duplicates or not, which is another dimension in the optimization of such operators, yet unrelated to the purpose of this analysis.

We only examine complete trees of outdegree 1 (simple lists), 2 and 3 (see Figure 4.5).

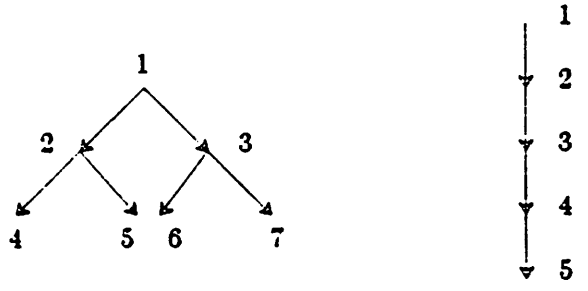

Figure 4.5. Complete trees of outdegree 2 and 1 (list).

The choice is made because these categories represent two extremes in terms of the ratio of depth over width of the tree. For the various parameters of the problem the following values are chosen:

$C = 7$	I/O cost to create a relation
$D = 10$	I/O cost to destroy a relation
$pgsz = 200$	Number of tuples fitting in a page
$buf = 50$	Number of buffers in the system

Choosing these values allows a comparison of the results of the analytical formulas above and the results obtained by simulating the semi-naive and smart algorithms on INGRES (see Section 4.2.3). The numbers for C, D, and buf have been taken as an average of many experiments. In the experiments, the tuple size has been 8-bytes and the page size 2K. Due to some overhead information in each page, $pgsz$ is 200. In Figure 4.6, the plots for the ratio $r = s\_naive\_io/smart\_io$ as a function of the depth of the list/tree, for lists and complete trees of outdegree 2 and 3 is shown.

Figure 4.6 validates the comments (a)-(d) above. As the relations grow, sorting becomes more significant and the semi-naive algorithm performs better compared to the smart one. However, it takes a considerably big relation for this to happen. For lists, a depth of at least 2048 is needed. Likewise, for complete trees of outdegree 2, the breakpoint depth is 16, which even though it is not a large number, it corresponds to a considerably large tree of 131072 edges (tuples). Incidently, this relation needs 1Mbyte of storage, whereas the result for A^* needs approximately 20Mbytes. The same is true for complete trees of outdegree 3. Figure 3.6c shows that the breakpoint is at depth 8. Thus, it takes an exceptionally large (deep/wide) relation for the semi-naive algorithm to perform less I/O than the smart one. Furthermore, the wider the tree is, the shorter it needs to be for this to happen.

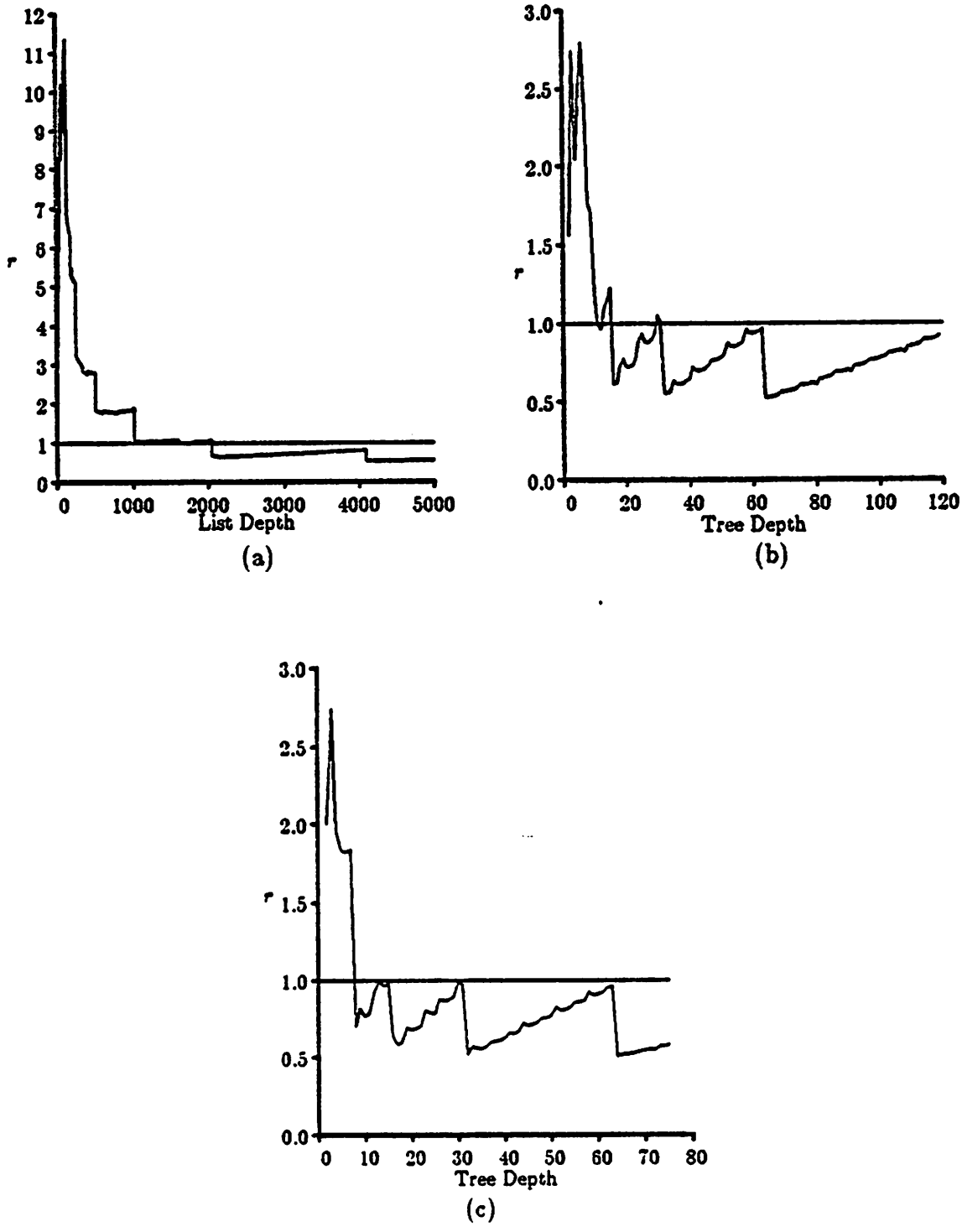

Figure 4.6. Expected relative I/O performance: $r = s\_naive\_io / smart\_io$.
(a) Lists, (b) Complete trees, outdegree 2, (c) Complete trees, outdegree 3

It is also very interesting to see the particular behavior of the ratio $r = s\_naive\_io/smart\_io$. In agreement with point (d) above, there are significant jumps in favor of the semi-naive algorithm at depths $N = 2^k$ for any k. This may be noticed at depths 256, 512, 1024, 2048, and 4096 for lists, or at depths 16, 32, and 64 for complete trees of outdegree 2 and 3. However, there is a noticeable difference in the behavior of lists and complete trees in this respect. For lists, r remains relatively invariant between A^{2^k} and $A^{2^{k+1}}$ for some k. On the contrary, for both cases of complete trees examined, as the amount of overcomputation decreases (approaching $A^{2^{k+1}-1}$) the relative performance of the smart algorithm improves significantly. For example, for a tree of outdegree 2 and N=16 there is a drop to $r = 0.61$, but as N grows it rises again to a point where for N=30 it is $r = 1.04$. Hence, $r = s\_naive\_io/smart\_io$ is not a monotone function of N.

It has been mentioned that the smart algorithm has been independently proposed in [Vald86], where its performance is analyzed in comparison with the semi-naive algorithm as well. It is difficult to accurately compare the results of this analysis with those in [Vald86]. We always use merge-scan join, whereas they mainly use a hash join similar to the one proposed in [DeWi84]. They also assume a very big buffer pool, whereas we do not. For both points above our constraints were put by the system used for the simulations (see Section 4.2.3). A common observation of both studies, however, is that the smart algorithm performs well in many cases.

4.2.3. Experimental Performance Results

We have simulated the semi-naive and smart algorithms using the commercial version of INGRES [RTI84] on a VAX[1] 11/780 running Unix[2] 4.3. The I/O costs observed for lists and complete trees of depth 2 and 3 are presented in Figure 4.7. The ratio $r = s\_naive\_io / smart\_io$ is shown again, together with the corresponding curve from the analysis of Section 4.2.2. Due to the space (and time) requirements of the experiments it has not been possible to compare the algorithms on very deep trees, so we have been unable to verify that after some point the semi-naive algorithm becomes better and identify that breakpoint. Nevertheless, to the extent that we did experiment, the results follow more or less the analysis of Section 4.2.2. Whenever there is a disagreement, it is attributed partly to the simulation overhead and partly to the pessimism of the analytical model about the way the optimizer uses the buffer pool and the cost of a sort.

The CPU time consumed by the algorithms in the experiments was also monitored. The monitored parameter was the ratio $r = s\_naive\_cpu / smart\_cpu$, where s_naive_cpu and $smart\_cpu$ are the CPU time consumed by the semi-naive and smart algorithms respectively. For the same categories of trees as above the observed r is shown in Figure 4.8. The smart algorithm is at least a factor of 2 better in performance for lists, whereas it is from marginally better to marginally worse for trees, for the depths examined. We speculate that, for trees of large depth, sorting cost is the significant factor, and hence the smart algorithm performs worse.

---

[1] VAX is a trademark of Digital Equipment Corporation.

[2] Unix is a trademark of Bell Laboratories.

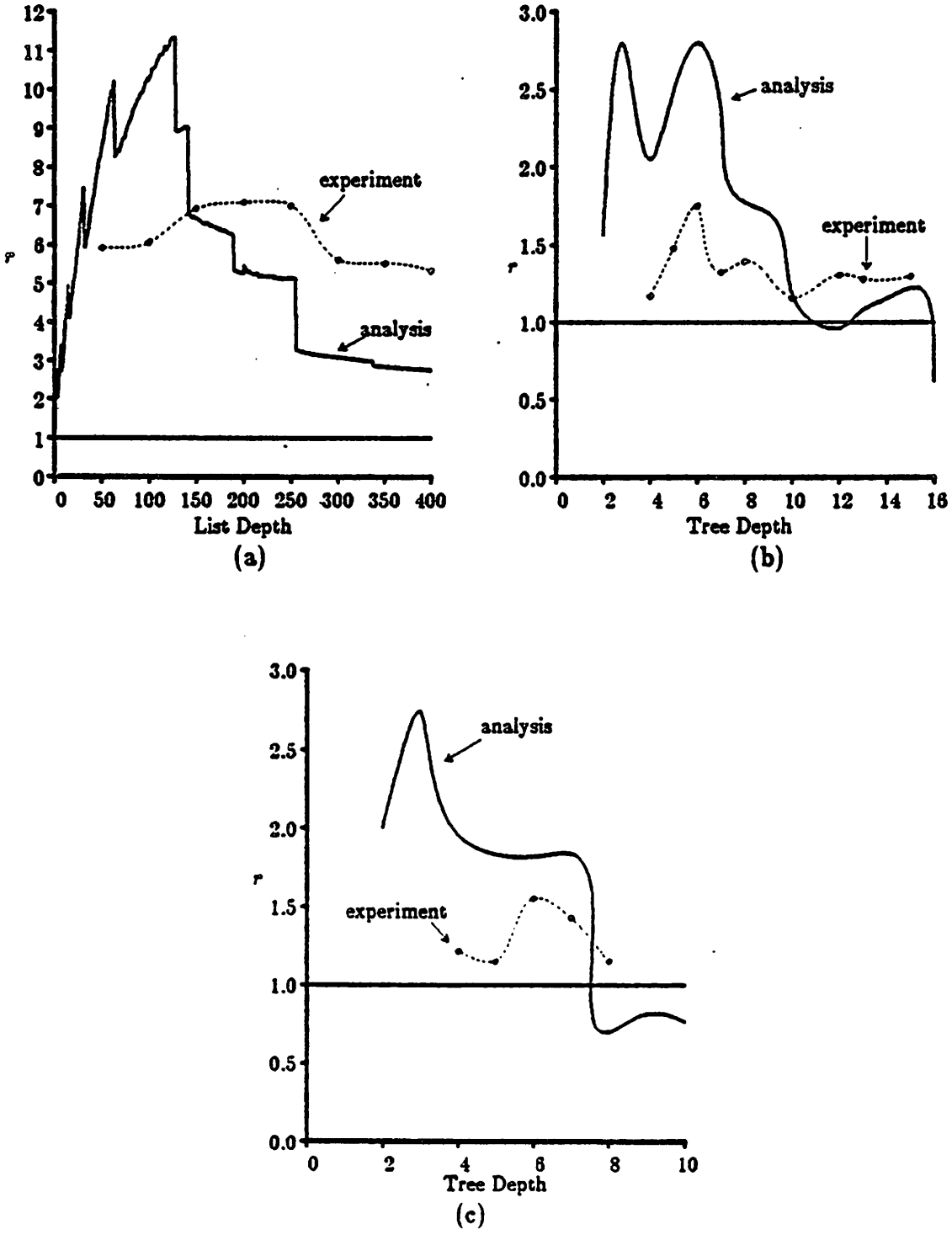

Figure 4.7. Observed relative I/O performance: $r = s\_naive\_io / smart\_io$.
(a) Lists, (b) Complete trees of outdegree 2, (c) Complete trees of outdegree 3

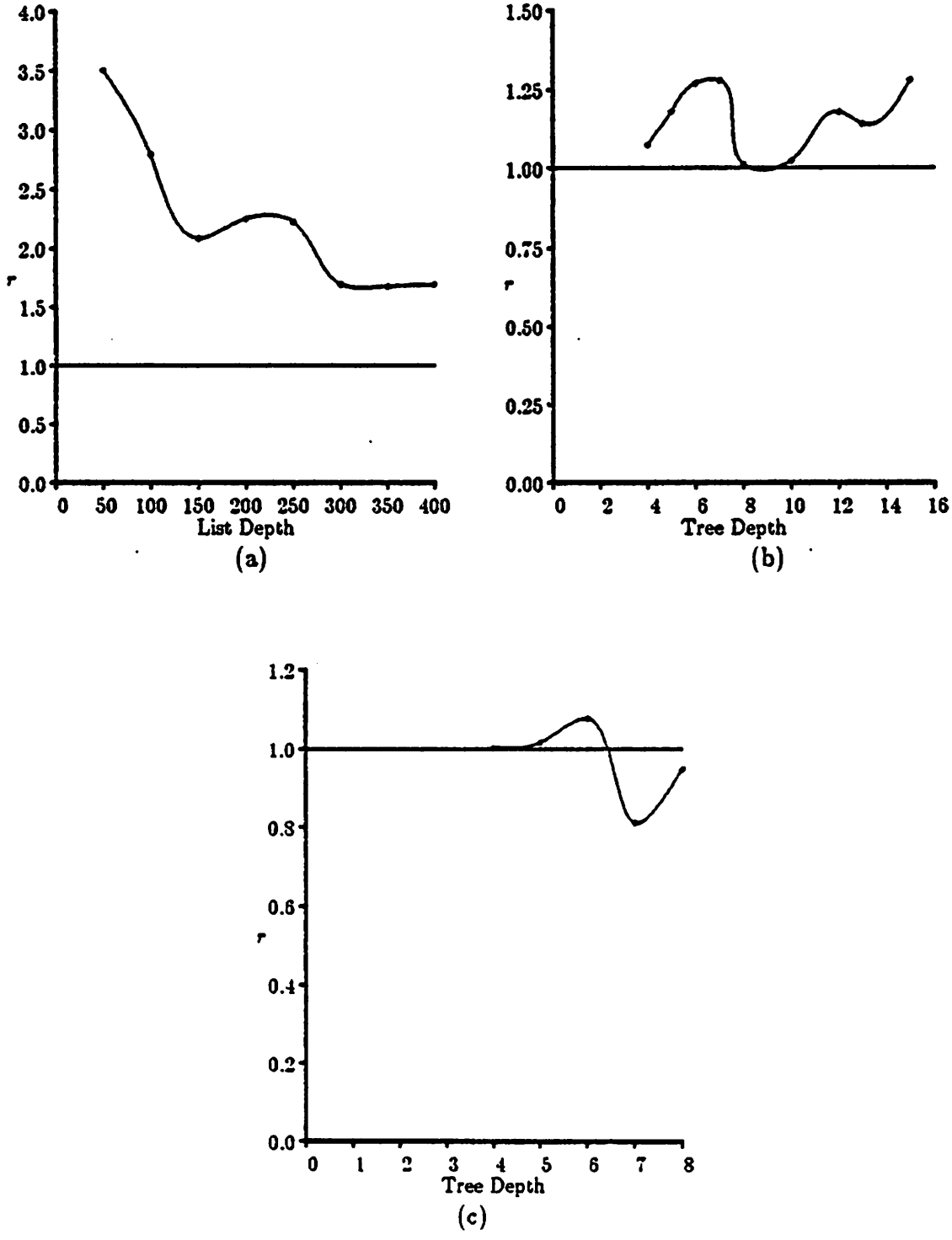

Figure 4.8. Observed relative CPU performance: $r = s\_naive\_cpu / smart\_cpu$.
(a) Lists, (b) Complete trees of outdegree 2, (c) Complete trees of outdegree 3

4.3. More Logarithmic Algorithms

A natural question that arises after looking at the smart algorithm (form (4.4)) is whether in the same spirit one can come up with other, occasionally even more efficient algorithms. The idea is to look at A^* as a regular expression for which there are many equivalent forms. Formulas (4.2), (4.3) and (4.4) represent only three of these forms and therefore correspond to only three of the possible algorithms to compute A^*. Some possible alternative algorithms can be realized by noticing that A^* may be written as

$$A^* = \cdots (1 + A^{12})(1 + A^6)(1 + A^3)(1 + A + A^2) \tag{4.5}$$

or

$$A^* = \prod_{k=0}^{\infty} (1 + A^{3^k} + A^{2 \cdot 3^k})$$

$$= \cdots (1 + A^9 + A^{18})(1 + A^3 + A^6)(1 + A + A^2). \tag{4.6}$$

Soon, one realizes that there is an infinite number of ways to write A^*, in the same sense that there is an infinite number of coding systems to code all integer numbers. Testing the performance of a large number of them is prohibitive because of their own time requirements and the complexity of their development. Good heuristics must be developed so that the large majority of the suboptimal candidate algorithms is not considered.

Expression (4.5) differs from (4.4) in that the grouping starts one iteration later, that is A and A^2 are computed separately, and then all the results are grouped together and A^3 is applied on them, and then everything is grouped again etc. On the other hand, expression (4.6) is even more aggressive in the grouping sense. It iterates

twice, before combining everything that it got up to that point and use it in the next pair of iterations.

In terms of the number of multiplications, expression (4.5) needs approximately $2\log_2 N$, where N is the number of multiplications of (4.3) (the greater N is, the closer the actual number of multiplications gets to $2\log_2 N$). In the same way, (4.6) needs about $3\log_3 N$ multiplications. Generalizing, algorithms can be created that need $m\log_m N$ multiplications, for arbitrary m, at the expense of making each multiplication more complex. However, with a few exceptions, $m = 2$ or $m = 3$ at most is all that is needed. In particular, assume that we concentrate on the general form

$$A^* = \prod_{k=0}^{\infty} \left(\sum_{l=0}^{m-1} A^{l*m^k} \right).$$

Formulas (4.4) and (4.6) are of this form for $m=2$ and $m=3$ respectively. The number of multiplications required by these formulas is approximately $m\log_m N$. An easy analysis shows that

$$m\log_m N \leq 2\log_2 N$$

only for m in $\{2,3,4\}$. In fact, the expression $m\log_m N$, with m restricted to the integers, has a minimum for $m=3$ independent of N. Therefore, all other options are more expensive than (4.4) and (4.6) and are considered no further.

Of all the alternative algorithms, we have experimented only with the one represented by expression (4.6), which is called the *minimal algorithm* (since it performs the minimum number of multiplications). The analysis in the previous paragraph is the dominant motive for this choice. Experimental results for lists and complete trees of outdegree 2 are shown in Figures 4.9a (I/O), 4.10a (CPU) and 4.9b (I/O),

4.10b (CPU) respectively. Assuming that the minimal algorithm consumed *minimal_io* and *minimal_cpu* units of I/O and CPU time respectively, we are interested in the I/O cost ratio $r = s\_naive\_io/minimal\_io$ and the CPU cost ratio $r = s\_naive\_cpu/minimal\_cpu$. In the figures below the curves for the corresponding ratios of the semi-naive over the smart algorithm are shown also, so that all three of them are compared simultaneously.

Figures 4.9 and 4.10 show that the minimal algorithm performs consistently better than the smart one in I/O (and even more so than the semi-naive algorithm). For the range of our experiments it does about a factor of 2 less I/O for lists, whereas for complete trees the analogous improvement is about a factor of 1.5. As it concerns CPU performance for lists the minimal is marginally better than the smart algorithm, whereas the opposite is the case for trees. A more extensive set of experimental results is definitely needed to get a better picture of the relative performance of all three algorithms.

In closing, a comment on the limited scope of the smart and minimal algorithms is appropriate. Minimizing the number of joins is an issue only when the complete A^* is computed. For queries that involve selections on the underlying relations it is the semi-naive algorithm that must be used, computing one power of A at a time and using the available selections at each point before performing the join. Most likely, this is faster than precomputing A^* using any algorithm and applying the selections afterwards. One may validly argue that queries involving selections are much more common than ones asking for a complete materialization of a recursively defined relation. Whether winning in these more rare cases is worth the implementation effort of

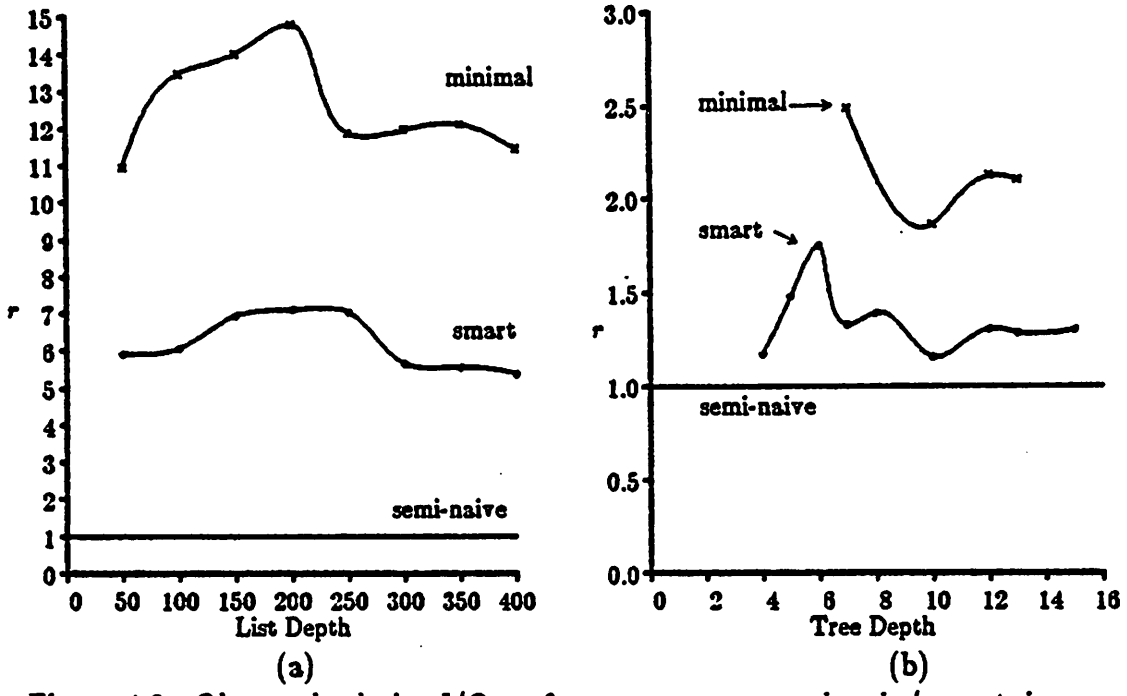

Figure 4.9. Observed relative I/O performance: $r = s\_naive\_io/smart\_io$ or $r = s\_naive\_io/minimal\_io$.
(a) Lists, (b) Complete trees of outdegree 2

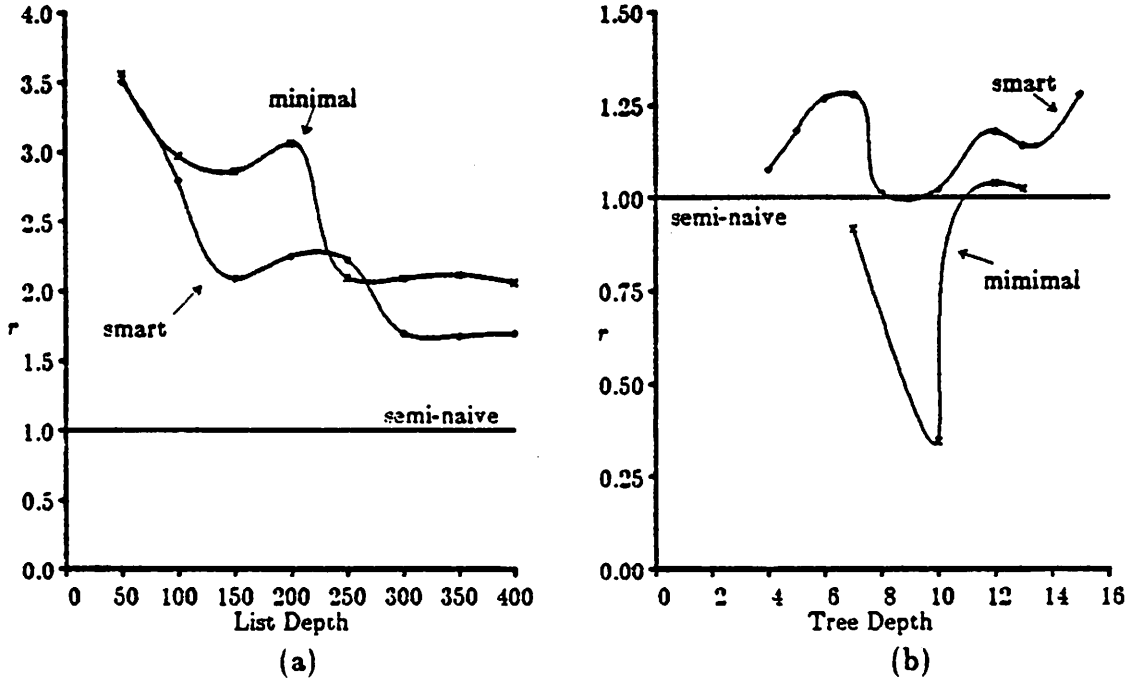

Figure 4.10. Observed relative CPU performance: $r = s\_naive\_cpu/smart\_cpu$ or $r = s\_naive\_cpu/minimal\_cpu$.
(a) Lists, (b) Complete trees of outdegree 2

the smart (or minimal) algorithm is questionable and to some extent application dependent.

4.4. Swapping

In Sections 4.2 and 4.3 a small number of equivalent expressions of the transitive closure A^* of a relational operator A have been identified. The main characteristic of these expressions is that A remains unchanged. Each expression is an alternative factorization of the polynomial $\sum_{k=0}^{\infty} A^k$.

One deviation from this, explored in [Ioan86], is to take advantage of the internal structure of A. That is, use the fact that A is some multiplication of more primitive operators (like join, project etc.) and, having these to be the units of algebraic manipulation, search for equivalent expressions representing more efficient algorithms. Consider an operator A such that $A = BC$. Then,

$$A^* = (BC)^* = 1 + B(CB)^*C. \tag{4.7}$$

Expression (4.7) represents a new algorithm for the computation of A^*, whose significant difference from the original one is the operator whose transitive closure is computed, namely $(CB)^*$ instead of $(BC)^*$. Depending on what B and C are and the contents of their parameter relations, the second algorithm may be more efficient. The algorithm realized by the transformation in (4.7) is called *swapping*.

No analytical or experimental studies of the performance of the swapping algorithm is available at this time. In compensation, the potential of the algorithm is illustrated by some characteristic examples.

Example 4.4: Consider the following linear recursive Horn clause:

$$\mathbf{P}(u,v,w) \wedge \mathbf{R}(u,v,w) \wedge \mathbf{S}(w,x,y,z) \rightarrow \mathbf{P}(x,y,z). \tag{4.8}$$

The operator A that corresponds to (4.8) is

$$A = (\mathbf{S} \bowtie)(\mathbf{R} \bowtie). \tag{4.9}$$

For presentation clarity the specific attributes on which the joins are performed have been omitted. In addition, it is assumed that after each join all the useless columns are projected out. Let $B = (\mathbf{S} \bowtie)$ and $C = (\mathbf{R} \bowtie)$ in (4.9). Swapping gives

$$A^* = 1 + B(CB)^*C = 1 + (\mathbf{S} \bowtie)\Big[(\mathbf{R} \bowtie)(\mathbf{S} \bowtie)\Big]^*(\mathbf{R} \bowtie).$$

The form above corresponds to the following set of Horn clauses:

$$\begin{gathered} \mathbf{P}(u,v,w) \wedge \mathbf{R}(u,v,w) \rightarrow \mathbf{P}'(w) \\ \mathbf{P}'(w) \wedge \mathbf{S}(w,r,s,t) \wedge \mathbf{R}(r,s,t) \rightarrow \mathbf{P}'(t) \\ \mathbf{P}'(t) \wedge \mathbf{S}(t,x,y,z) \rightarrow \mathbf{P}(x,y,z). \end{gathered} \tag{4.10}$$

This system gives the same answer to any query on $\mathbf{P}$ as the original Horn clause. However using (4.10) may be more efficient. Assuming that most of the processing time is consumed in the recursive Horn clause, it may well be that the second Horn clause of (4.10) is more efficient than (4.8). This depends entirely on the sizes of the relations involved and the sizes of the intermediate results of operators applied on these relations. Note, that (4.10) is only one of possible systems equivalent to (4.8) that swapping may create, each one of which has a different recursive Horn clause. □

Example 4.5: Consider an operator A for which $A = B\pi$, i.e. A is a product of some operator B with a projection π. The nice property of A is that, with the appropriate application of swapping, A^* is replaced by the transitive closure of an

operator whose domain and range are relations of fewer columns than A. This is evident in (4.11):

$$A^* = 1 + B(\pi B)^*\pi. \tag{4.11}$$

The new algorithm is guaranteed to be more efficient since it manipulates smaller relations.

A Horn clause for which the above hold is

$$\mathbf{P}(w,z) \wedge \mathbf{R}(z,y) \wedge \mathbf{S}(x) \rightarrow \mathbf{P}(x,y).$$

Since the variable w does not appear anywhere in the Horn clause except under **P** in the qualification (implying a projection on the incoming relation), swapping may be applied to give

$$\mathbf{P}(w,z) \rightarrow \mathbf{P}'(z)$$

$$\mathbf{P}'(z) \wedge \mathbf{R}(z,y) \wedge \mathbf{S}(x) \rightarrow \mathbf{P}'(y)$$

$$\mathbf{P}'(z) \wedge \mathbf{R}(z,y) \wedge \mathbf{S}(x) \rightarrow \mathbf{P}(x,y).$$

In this particular case, a test whether **S** is empty or not can be performed right from the beginning. This simplifies the above system of Horn clauses even more, since the second one can be replaced by

$$\mathbf{P}'(z) \wedge \mathbf{R}(z,y) \rightarrow \mathbf{P}'(y). \qquad \square$$

A final comment for the computation of A^* is that the smart or the minimal algorithm can be combined with the swapping algorithm for the final execution plan. Swapping can also be combined with all of the techniques described in Section 4.1 for taking into account the possible selections of a given query to speed up processing. It is evident that any sophisticated optimizer for recursive queries has a problem of a big

search space size. In the future good heuristics should be developed to make the search space manageable. This is further discussed in Chapter 5.

4.5. Summary

Having the answer to a recursive inference process in an explicit form of a relational operator allows the use of the manipulative power of the closed semiring, where all such operators have been embedded. This way, new algorithms that are potentially more efficient than the ones commonly used are realized. Two such algorithms, called smart and minimal, have been proposed for the computation of the transitive closure of a relational operator. Their common characteristic is that they perform a smaller number of operator multiplications on larger relations than the semi-naive algorithm does. Analytical study as well as experimental measurements shows that as the relations increase in size the new algorithms are less effective. Nevertheless, the point after which the new algorithms cease to be optimal corresponds to considerably large relations. Thus, for a wide range of relation sizes the new algorithms are shown to be cost-effective.

CHAPTER 5

OPTIMIZATION ALGORITHMS

The performance comparison of the semi-naive, smart, and minimal algorithms in Chapter 4 shows that none of them is universally optimal, for all database instances. Naturally the problem of choosing each time the one that performs the best arises. This optimization problem has never been addressed before in an environment with recursively defined relations. In addition, we will propose two (heuristic) algorithms to both explore some space of strategies to compute A^* and find the one with the least expected execution cost.

5.1. Strategy Space

Consider a linear operator A and its transitive closure A^*. A *strategy* to compute A^* is a sequence of multiplications and additions of simpler algebraic expressions (operators) forming A^*. Thus, the set of all the algebraicly equivalent forms of A^* is the strategy space considered. The goal is to find the cheapest one for the database concerned.

Even though the relative cost of two strategies is in general database dependent there is a limited number of cases for which the suboptimality of a strategy is provable by purely syntactic means, i.e. independent of the database. Bancilhon has proposed that, when a strategy is not *duplicate-free*, i.e. some operator is applied on some set of tuples more than once (or in the algebraic framework two operators are multi-

plied with each other more than once), then it is not optimal [Banc85]. Using this criterion, he has shown that $(1 + A)^N$, which corresponds to the naive evaluation, is not duplicate-free and, therefore, not cost effective. This was pointed out in Section 4.1.1 also. Bancilhon's suboptimality criterion is extended as follows:

Definition 5.1: A strategy is *repetition-free* if it does not form any algebraic expression more than once.

Notice that a duplicate-free strategy is repetition-free also. However the reverse is not true. For example, producing both $A(AA)$ and $(AA)A$ in a strategy is not repetition-free even though it is duplicate-free. Clearly, a strategy that is not repetition-free is not cost effective since it includes redundant computation. Identifying non-repetition-free strategies and removing them from the strategy space considered is highly desirable. Theorem 5.1 provides a result in this direction.

Theorem 5.1: Consider two algebraic expressions B and C such that $B = A^n$ and $C = \sum_{i=1}^{m} A^{k_i}$. Consider a strategy that involves the multiplication BC. If $n \in \{k_j - k_l : 1 \leq j,l \leq m\}$ then the strategy is not repetition-free.

Proof: Let $B = A^n$ with $n = k_j - k_l$, for some $1 \leq j,l \leq m$. Multiplying B with C produces the sum of all the powers of A of the form A^{n+k_i}, $1 \leq i \leq m$. For $i = l$ the operator $A^{n+k_l} = A^{k_j - k_l + k_l} = A^{k_j}$ is produced, which has already been formed before as part of C. Therefore, multiplying B and C prohibits the strategy from being repetition-free. □

Example 5.1: Consider formula (4.2) which corresponds to the naive evaluation:

$$A^* = \lim_{k\to\infty}(1+A)^k = \cdots(1+A)(1+A).$$

Its inefficiency follows directly from Theorem 5.1. Naive evaluation is not repetition-free, because A is multiplied with $(1+A)$, which corresponds to the values n=1, k_j=0 and k_l=1 in the statement of the theorem. □

Even though there is a unique algebraic expression equal to A^* associated with each evaluation of A^*, the opposite does not hold. There is a certain algorithmic information lost when an evaluation (a strategy) is "flattened out" to an algebraic expression. In particular, whether a repeated subexpression is computed only once or not cannot be decided from the algebraic expression alone. For example, consider the expression $A^2 + (A^2)A$. There is no indication of whether A^2 is computed only once or twice. For this reason, a strategy is represented by a directed acyclic graph, whose leaves are the primitive operators involved in the computation of A^*, and the other nodes are multiplications and additions applied on their children. In the examples that follow, the edges of the graphs always have top to bottom direction.

Example 5.2: The graphs in Figure 5.1 represent the states $\sum_{k=0}^{2} A^k$ and $(1+A)^2$ respectively. Also, the algebraic expression $A^2 + (A^2)A$ may correspond to any of the graphs in Figure 5.2, depending on whether A^2 is computed once or twice.

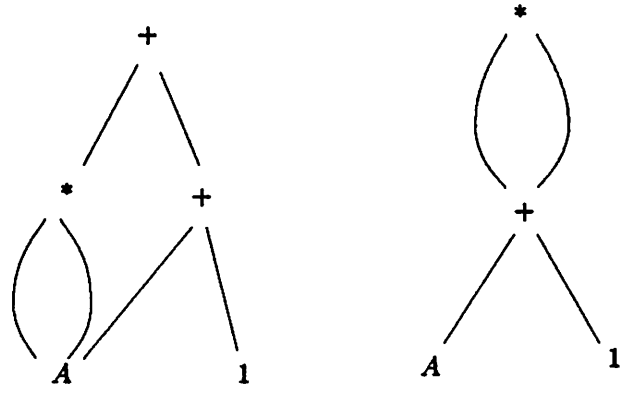

Figure 5.1. Strategies corresponding to $1 + A + A^2$ and $(1 + A)^2$.

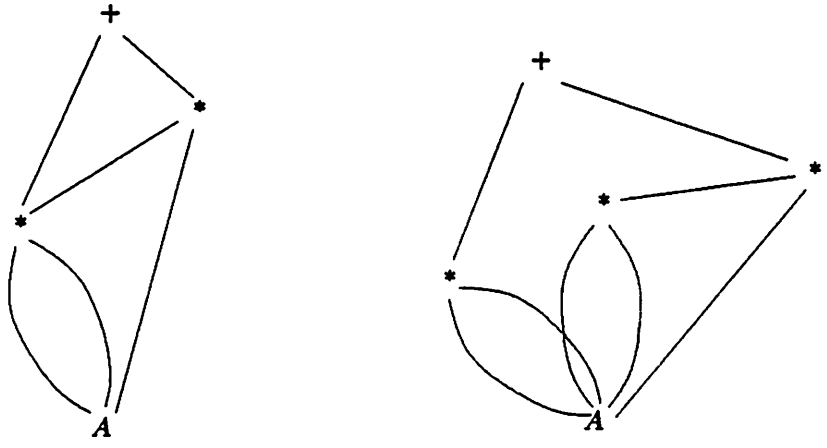

Figure 5.2. Two different strategies corresponding to $A^2 + (A^2)A$. □

Definition 5.2: The *depth* of a strategy is the depth of its corresponding graph, i.e. the maximum path-length in the graph.

Notice that all the graphs corresponding to a single algebraic expression have the same depth. Hence, depth is well defined for an algebraic expression also.

Example 5.3: The algebraic expression $1 + A$ has depth 1, whereas $AB + CD$ has depth 2 (see Figure 5.3).

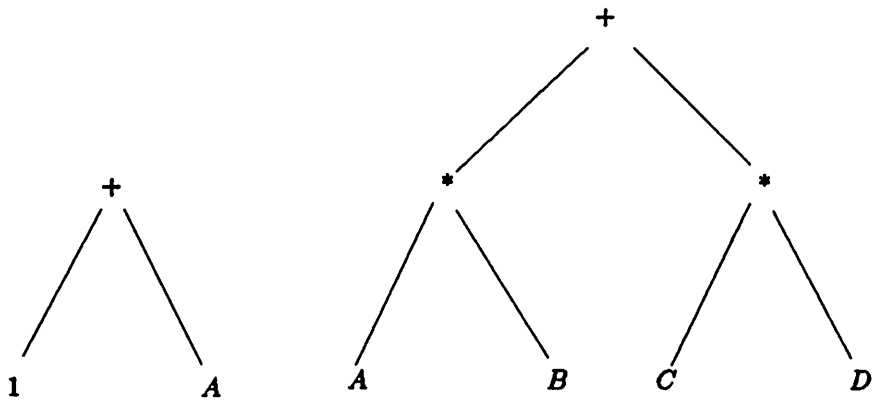

Figure 5.3. $1 + A$ of depth 1 and $AB + CD$ of depth 2. □

For the remainder of the chapter, a strategy is identified with its graph. Occasionally, if this does not cause any confusion, the corresponding algebraic expression is used.

In the following two sections, two optimization algorithms for the computation of A^* are described. Whenever possible, Theorem 5.1 and other similar results are applied to exclude some of the strategies that are known to be suboptimal. The first such algorithm is a *simulated annealing* process whereas the second one is based on the *dynamic programming* principle.

5.2. Optimization by Simulated Annealing

5.2.1. Simulated Annealing

Simulated annealing is a *Monte Carlo* optimization technique proposed by Kirkpatrick et al. for complex problems that involve many degrees of freedom [Kirk83]. Such problems are modeled by a state space, each state corresponding to a solution to the problem. A cost is associated with each state, and the goal is to find the state that has the globally minimum cost associated with it. For complex problems with very large state space exhaustive exploration of all the states is impractical.

Probabilistic hill climbing algorithms, like simulated annealing, attempt to find the global minimum by (hopefully) traversing only part of the state space. They move from state to state allowing both downhill and uphill moves, i.e. moves that reduce and moves that increase the cost of the state respectively. The purpose of the latter kind is to allow the algorithm to escape from local minima it may occasionally encounter. For example, consider the one dimensional function of Figure 5.4. States S_1 and S_2 are local minima, whereas S_3 is the global minimum. A probabilistic hill climbing algorithm works in such a way that, even if at any point finds itself in state S_2, it climbs up again to eventually terminate in S_3.

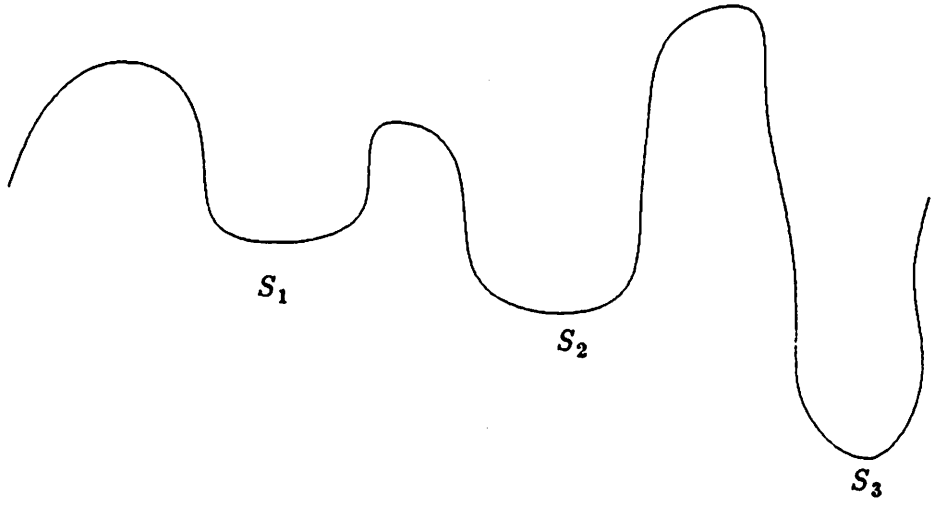

Figure 5.4. Local and global minima.

In simulated annealing the uphill moves are controlled by a parameter T, the temperature. The higher T is the higher the probability an uphill move is taken. As time passes T decreases, and at the end, when the system is "frozen" (T very close to 0), the probability of making an uphill move is negligible. This simulates the annealing process of growing a crystal in a fluid by melting the fluid (high T) and then

slowly decreasing T until the crystal is formed. The fluid is at a low energy state. In optimizing by simulated annealing, the cost function plays the role of the energy in the physical phenomenon.

Simulated annealing works as follows: Consider a state space S_A (A for Annealing), and a function N_A: $S_A \rightarrow Po(S_A)$, such that for a state s, $N_A(s)$ is the set of neighbors of s in S_A. Also, consider a cost function c_A: $S \rightarrow \mathbb{R}$ that associates a cost with each state in S. Visualizing the state space as the set of nodes in a directed graph, $N_A(s)$ represents the edges emanating from the state (node) s. The state space is assumed to be strongly connected, i.e. there exists a path from any node to any other node. Algorithm 5.1 shows the basic structure of simulated annealing.

```
s = s0;
T = T0;
while (not_yet_frozen) do
    while (not_yet_in_equilibrium) do
        s' = random state in NA(s);                          (step 1)
        Δc = cA(s')-cA(s);
        if (Δc ≤0) then s = s';
        if (Δc >0) then s = s' with probability e^(-Δc/T);   (step 2)
    T = reduce (T);
return(s);
```

Algorithm 5.1. Simulated Annealing.

There are two major loops in Algorithm 5.1. In the inner loop, the temperature T is kept constant as the algorithm explores part of the state space. Downhill moves are always accepted but uphill moves are accepted with some probability less than 1. After some form of equilibrium is reached, the temperature is reduced according to some function (*reduce*) and the inner loop is entered again. The whole process stops when the freezing point is reached. Each iteration of the outer loop, which is done at

a constant temperature, is called a *stage*.

There have been many theoretical investigations for the behavior of the algorithm [Rome84,Rome85,Haje85]. It has been shown that, under certain conditions satisfied by the way the next state is "randomly" chosen (step 1 in Algorithm 5.1) and the way the temperature is reduced (step 2 in Algorithm 5.1), as the time approaches ∞ the algorithm converges to a state s such that $c_A(s)$ is a global minimum of c_A. Hence, the original optimization goal is achieved.

5.2.2. Simulated Annealing for A^*

Simulated annealing has been applied to a great variety of optimization problems often with substantial success. The major field of its application seems to be VLSI design, in particular standard cell placement and global routing [Rome84,Sech86b], which was also identified as a potential application in the original proposal of simulated annealing as an optimization technique [Kirk83]. However, simulated annealing has been applied to problems of other areas also (e.g. pattern recognition [Ackl85]). There have also been experimental studies with simulated annealing applied on more traditional optimization problems, in particular graph partitioning, the traveling salesman problem, number partitioning and graph coloring [Arag84].

The successful application of simulated annealing on this great variety of optimization problems together with its theoretical foundation and its elegant simplicity has been the primary motivation to devise a simulated annealing algorithm for the optimization of the computation of A^*, for some linear relational operator A. Even though the structure of the simulated annealing algorithm is problem-independent, there are

some particular parameters that are specific to the problem. These are the state space S_A to be explored, the neighbor's set for each state s in S_A (given by the function $N_A(s)$) and the cost function c_A. The definitions of these parameters for the specific problem concerned are given in this section. In addition to the above, there are some parameters of the algorithm that are implementation-dependent (and somewhat problem-dependent also) and they are specified in the next section, which discusses the implementation of the algorithm.

- **State space**

Every strategy that computes A^* is a state in the state space S_A to be explored. According to Section 5.1, a strategy is a graph. Therefore, the state space is a graph of graphs. Each state corresponds to an algebraic expression equal to A^*. Since, for any specific computation, A^* is a finite operator, only finite sums of the form $\sum_{k=0}^{N} A^k$ are considered. For example, $\sum_{k=0}^{N} A^k$ and $(1 + A)^N$ are two distinct states in the state space.

- **Neighboring states**

For each state $s \in S_A$ the set of its neighbors $N_A(s)$ is determined by the properties of multiplication and addition within the closed semiring E_R. In particular, s' is a neighbor of s (i.e. $s' \in N_A(s)$), if s' can be produced by applying one of the following transformations to a *single* node in the graph of s.

Associativity of $+$: For three operators A, B and C, $(A + B) + C = A + (B + C)$ (see Figure 5.5).

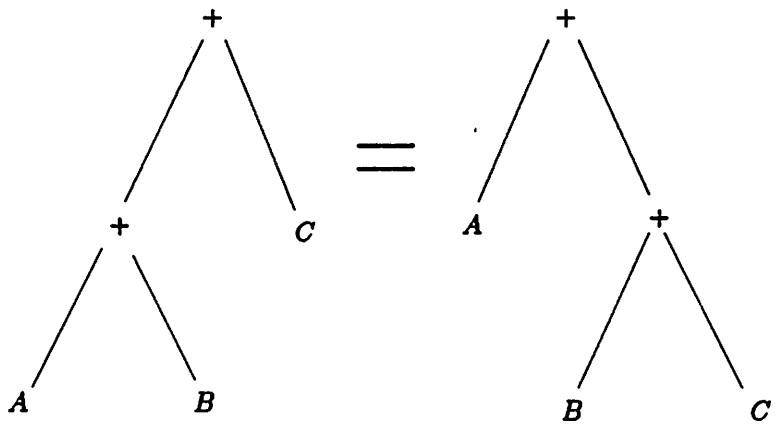

Figure 5.5. State transformation by associativity of +.

Associativity of $*$: For three operators A, B and C, $(A * B) * C = A * (B * C)$ (see Figure 5.6).

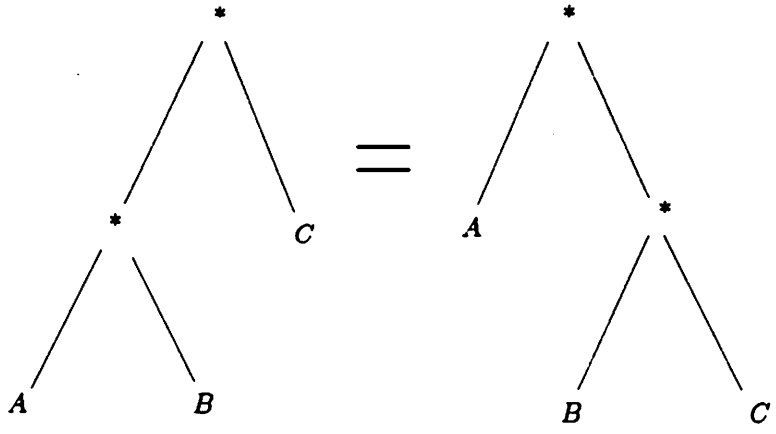

Figure 5.6. State transformation by associativity of $*$.

Commutativity of +: For two operators A and B, $A + B = B + A$ (see Figure 5.7).

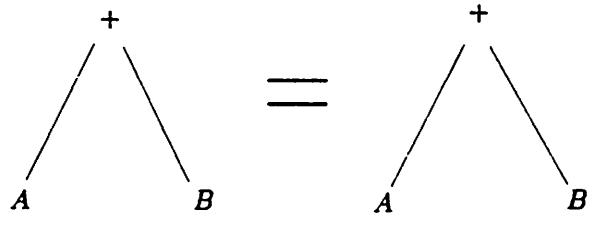

Figure 5.7. State transformation by commutativity of +.

*Distributivity of * over +*: For three operators A, B and C, $A(B + C) = AB + AC$ (see Figure 5.8) and $(B + C)A = BA + CA$.

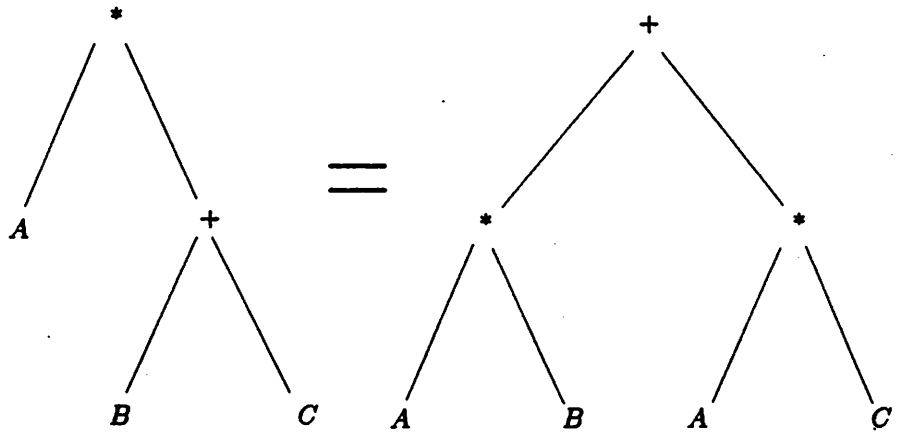

Figure 5.8. State transformation by distributivity of * over +.

*Distributivity of * over + with 1 the multiplicative identity*: For two operators A and B, $A(B + 1) = AB + A$ (see Figure 5.9) and $(B + 1)A = BA + A$.

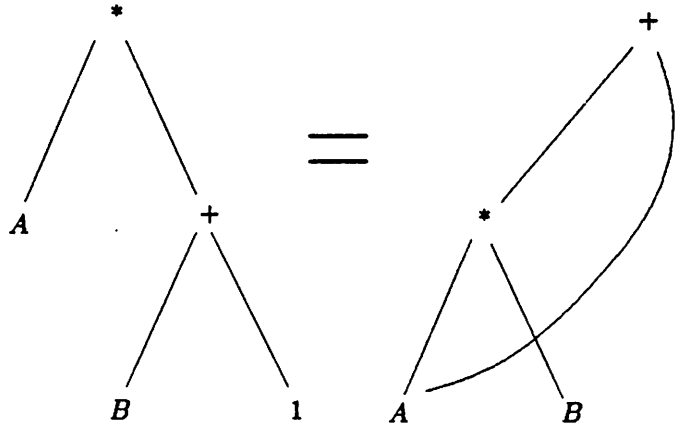

Figure 5.9. State transformation by distributivity of * over + with 1 the multiplicative identity.

Each transformation corresponds to some property in the first three parts of the closed semiring definition (Definition 2.1). There are a few notable exceptions to that. Namely, there are no transformations corresponding to the properties that 0 is the additive identity $(A + 0 = 0 + A = A)$, 1 is the multiplicative identity

($A * 1 = 1 * A = A$), 0 is an annihilator ($A * 0 = 0 * A = 0$), and + is idempotent ($A + A = A$). The first three are excluded because they create strategies that are equivalent to strategies already in S_A. The last one is excluded because it creates strategies (states) that are not repetition-free. Its exclusion reduces the state space size by removing states that are known to be non-optimal. For example, the state $(1 + A)^N$ is not considered, since the idempotency of + is not a transformation (e.g. $(1 + A)^2 = 1 + A + A + A^2$). The state space can be shrunk further, by applying Theorem 5.1 or any other similar result about ordering operators according to their costs. However, any result of this form can be applied only if the reduced state space remains strongly connected. Otherwise, the optimal state may not be reachable from the initial state .

• **Cost function**

Since the problem addressed is one of database query optimization, the cost function c_A is the cost of applying the individual operators to their input relations. In a real system, this is an estimate produced from the statistics that the system keeps about the database. For the purpose of the presentation of the algorithm we assume that c_A is given.

Modeling the cost c_A of applying relational operators on relations may be done at many levels of detail. In this sense the choice of the cost function c_A is implementation-dependent also. Producing an accurate model is a difficult problem, even for the case of regular query optimization [Wong76, Seli79, Jark84, Mack86], and a whole research area of its own. Since it does not affect the applicability or the perfor-

mance of the simulated annealing algorithm, our description remains general and assumes that c_A is given. In fact, c_A could be an arbitrary cost function, completely unrelated to relational operator costs. The optimization problem would still be well defined by the state space S_A and the neighbors of each state $N_A(s)$ and simulated annealing would be applicable as well.

5.2.3. Implementation of Simulated Annealing for A^*

We have implemented simulated annealing for the optimization of A^* using the state space, neighbors and cost function presented in Section 5.2.2 (for the cost function, a simple model was used, which is described shortly). The implementation was done in Franz LISP [Wile83] under the Unix 4.3 operating system on a VAX 11/780. LISP was chosen over C, which would be the other obvious choice in our environment, because of the graph form of the states and LISP's ability to manage lists (and therefore graphs) efficiently and elegantly. The cost function c_A is a simple model of the I/O cost of database operations. Join and union are the only operations modeled. The cost of a join is the product of the sizes of the two relations plus some additional linear terms to read the relations and write the result. The cost of a union is the sum of the sizes of the two relations to read them plus the size of the result to write it out.

It has been mentioned already that there are some parameters to the simulated annealing algorithm that are implementation-dependent. These are the initial state s_0, the initial temperature T_0, the freezing criterion, the equilibrium criterion, the way the next state is chosen randomly, and the way the temperature is reduced from stage to stage (the *reduce* routine of Algorithm 5.1). In the current implementation they

have been chosen as follows:

• **Initial state s_0**

From the theoretical analysis of the simulated annealing algorithm, it is derived that the effectiveness of the algorithm in finding the global minimum state is independent of the choice of the initial state of the execution [Rome84,Rome85,Haje85]. Moreover, this is verified by many experimental studies with simulating annealing algorithms [Arag84,Sech86b]. Since any initial state is as good as any other, the one corresponding to the semi-naive evaluation is chosen for this purpose. For example, assuming that only the first two powers of A are to be computed, that is $1 + A + A^2$, the initial state is shown in Figure 5.10.

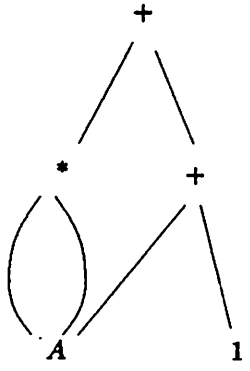

Figure 5.10. Initial state to compute $1 + A + A^2$.

• **Initial temperature**

Contrary to what is the case with the initial state, choosing the right initial temperature T_0 is important. It has to be considerably high so that the system is relatively "hot" in the beginning and allows many uphill moves to be accepted. For the range of experiments performed, the initial temperature was chosen to be twice the cost of the initial state:

$$T_0 = 2c_A(s_0).$$

• **Freezing criterion**

There is a great variety of freezing criteria proposed in the literature. Most of them are a combination of tests verifying that the system is at a low temperature and that it does not change state often, i.e. it has converged to its final state. For this implementation, the criterion used is a combination of the ones given in [Arag84] and [Sech86b] and consists of two parts. First, the temperature has to be below 1 ($T \leq 1$) and second, the value of c_A at the final state has to be the same for four consecutive stages.

• **Equilibrium criterion**

In all the implementations of simulated annealing that we are aware of, every stage consists of a specific number of iterations through the inner loop. This number may be independent of the temperature of the stage [Arag84], or it may get larger as the temperature decreases [Sech86b] (in the latter reference this is done indirectly through considering only a smaller subspace of the original state space). The advantage of having more state transitions during the later stages of the execution has been theoretically justified as well [Mitr85]. We have chosen to have a constant number of iterations through the inner loop, independent of the temperature. This number is equal to *epoch_factor***epoch*, where *epoch_factor* is an arbitrary factor (in the experiments chosen to be 16), and *epoch* is the number of neighbors the initial solution has (both the terminology and formulation is from [Arag84]).

• **Choosing the next transition**

At any point in the execution, the next state is chosen randomly from among the neighbors of the current state. More specifically, suppose that there is a transition probability matrix R over the state space S, such that

$$R(s,s') = \begin{cases} \frac{1}{|N_A(s)|} & \text{if } s' \in N_A(s) \\ 0 & \text{otherwise} \end{cases}.$$

Recall that $N(s)$ is the set of neighbors of state s. The probability that an attempt is made to move from s to s' is equal to $R(s,s')$.

• **Reducing the temperature T**

Many *cooling schedules* have been proposed for the simulated annealing process. We distinguish two of them. Specifically, Hajek [Haje85] proposes reducing the temperature according to the formula

$$T_k = \frac{d}{log(k+1)}.$$

In the above, k is the current stage number (it represents time also) and d is some constant for which he gives a sufficient value for the algorithm to converge to the global minimum. Unfortunately, from a practical point of view this is not a desirable schedule, because it is very slow. For this reason, another schedule has been proposed that reduces the temperature according to the formula

$$T_{new} = \alpha(T_{old})T_{old}.$$

The factor α is a number between 0 and 1. In [Rome84,Sech86b] α ranges over time (that is, it depends on T_{old}). It is smaller in the beginning (cooling the system fast),

then it rises up to higher values (slowing down the cooling process), and eventually it becomes small again to drive the system down to a minimum without any uphill moves. On the contrary, in [Arag84] it is suggested that α does not change but remains constant at a relatively high value in the range of 0.9 to 0.95. We have experimented with both a constant α=0.95 and a variant α modified according to Table 5.1.

$T_0/T \leq$	α
2	0.80
4	0.85
8	0.90
∞	0.95

Table 5.1. Factor to reduce the temperature

5.2.4. Experiments with Simulated Annealing

We have performed a limited number of experiments with the system. We have applied it to small examples with three and seven terms of A^*, i.e. $\sum_{k=0}^{n} A^k$, n=3 or n=7. In all the experiments, A represented transitive closure of a simple list, as in the first set of experiments described in Section 4.2. Notice that this represents a very unfavorable situation for simulated annealing because the state space is not very big, A is not complex, and the sizes of the relations are so small that one cannot expect significant variations in the cost of the various states. Nevertheless, even though it is premature to draw any general conclusions from such a limited range of experiments, we may dare say that the results have been encouraging. For the case with n=3, the algorithm always found the minimum cost state, which incidentally is the smart algo-

rithm. For the case with $n=7$, even though it did not always find the global minimum, it did converge to a state with a cost close to it and much smaller than the cost of the original state. Further experimentation is definitely needed to verify the general applicability of the algorithm to the problem.

There are a few points in the way we have applied simulated annealing to the problem where there is room for improvement. The first one is that $|N_A(s)|$ is small for any state s. That is, the transformations described in Section 5.2.2 for changing states give a relatively small number of neighbors to each state. This has the implication that the paths between states tend to be long, and it takes many steps for the algorithm to traverse them. A possible solution is to allow the algorithm to perform a number of these transformations at once (as a single move), thereby putting direct transitions among more states than now. We expect this to reduce the running time of the algorithm significantly.

The solution proposed to the first problem above gracefully affects the second one also. That is, the cost change (Δc in algorithm 5.1) when making a transition is usually small relatively to the total cost. For example, changing the order of two joins (by applying associativity of multiplication) cannot significantly affect the total cost of the whole computation of A^*, if the latter contains many terms. Hence, even at very high temperatures the system does not undergo drastic changes in terms of its cost. It has been experimentally verified that a small range of Δc is most of the time disastrous for the effectiveness of the algorithm [Sech86a]. The combination of a number of transformations in one state transition may give the desired range to Δc and have the algorithm perform better. Another possibility may be to choose c_A such that it is a

monotone function of the actual cost of applying operators on relations but has a better behavior in the sense mentioned above.

Finally, the way the next state is chosen in the current implementation causes problems also. Many of the transformations applied on the state leave its cost unchanged. For example, applying commutativity of + leaves the cost of the state unchanged. This tends to create a number of *plateaux* in the state space. The system tends to wander in these plateaux for a long time without any progress, neither going downhill, nor going uphill hoping to explore a more fruitful area of the state space later. Since every applicable transformation is equally likely to happen, the majority of the transitions are of this form, with no change in the cost. A better procedure is needed for choosing the next state, giving higher probability to transitions that do affect the cost. This discriminative treatment of neighbors has been used elsewhere with great effectiveness [Sech86b].

5.3. Optimization by Dynamic Programming

Simulated annealing is a probabilistic algorithm to find the most efficient way to compute A^*. We now describe a deterministic algorithm for the same problem. It is based on the dynamic programming principle [Lars78] and uses the algebraic formulation of recursion of Chapter 2. Regular query optimization has been modeled successfully as a dynamic programming process [Lafo85]. The idea is extended here for the case of recursively defined relations. The task is fundamentally more difficult, because the number of alternative ways to answer a query is much larger.

5.3.1. Dynamic Programming for A^*

It is assumed that the dynamic programming principle is widely known. So, no description of the algorithm is given. We only specify the problem-dependent parameters of the algorithm. Dynamic programming is applied on a state space S_D (D for Dynamic Programming). The neighbors of each state s in S_D are given by the function $N_D(s)$: $S_D \rightarrow Po(S_D)$. Moreover, there is a cost associated with each transition, given by the function c_D: $S_D X S_D \rightarrow \mathbb{R}$. These are the three problem-dependent parameters that have to be specified for the particular problem concerned.

Consider an operator of the form $A = A_1 A_2 \cdots A_l$ with A_i, $1 \leq i \leq l$ primitive relational operators. Given a specific A corresponding to a single Horn clause, there are various forms it can take as a product of simpler operators, at various levels of detail. The greater l is the more details about the internal structure of A are known and the larger the strategy space explored. The internal structure of A is ignored by choosing $l=1$. By doing so, certain strategies are excluded, e.g. the swapping algorithm of Section 4.4. The specific decomposition of A does not affect the way dynamic programming works. Thus, the approach taken here is flexible.

• **State space**

Each state in S_D contains some algebraic expressions. Definition 5.2 associates each algebraic expression with a unique integer, its depth. Using the notion of depth, an index k, $k \geq 0$, is associated with every state as follows:

Definition 5.3: Consider a state s. The *index* k of s is the maximum depth associated with any algebraic expression in s is k.

The state space S_D is defined as follows:

- $s = \{1, A_1, A_2, \ldots, A_l\}$ is a state in S_D. It is of index 0 and is called the *root*.
- Let s be a state in S_D of index k. Let $s[k]$ be the subset of algebraic expressions of s that are of depth exactly k (there is at least one of those, because otherwise s would not be of index k). In other words s is of the form $s = s' \cup s[k]$, with $s' \cap s[k] = \emptyset$. Then $s \cup f$ is a state in S_D of index $k+1$, where

$$f \subseteq \{B_i + B_j : B_i \in s[k],\ B_j \in s, \text{ and } B_i, B_j \text{ are domain- and range-compatible}\}$$
$$\cup \{B_i * B_j : B_i \in s[k],\ B_j \in s, \text{ and } B_i \text{ is dr-compatible to } B_j\}$$
$$\cup \{B_i * B_j : B_i \in s,\ B_j \in s[k], \text{ and } B_i \text{ is dr-compatible to } B_j\}.$$

- Nothing else is a state in S_D.

• Neighboring states

There is a transition from a state s to a state s' (i.e. $s' \in N_D(s)$), if s is of index k, s' is of index $k+1$ and $s \subseteq s'$. Hence, in the previous paragraph, the state space was defined by constructing the children of a state from the father. It takes an easy induction on the state index to show that the state space is a tree. Apparently, the set of algebraic expressions in each state is a subset of the one in any of its children (and by transitivity any of its descendents).

• Cost function

There is a cost function c_D defined on the transitions in the state space S_D. Each transition from one state to another involves a number of multiplications or additions of operators. Naturally, the sum of the costs of these operations is the cost associated with the transition. Again, choosing the right cost function so that it

adequately models the real cost of the operations is up to the implementor. For the purpose of this presentation we assume that c_D is given. The cost of a path is accordingly defined as the sum of the costs of the edges (transitions) along the path.

It has been stated that, for every specific database instance, A^* is equal to a finite number of terms. The state space constructed for a finite sum is finite. Moreover it is a tree, whose leaves contain A^* (the equivalent finite sum) as one of their algebraic expressions, each one being constructed in a different way. The goal is to find the one that has the minimum cost associated with the path from the root to it.

Example 5.4: Consider optimizing $1 + A + A^2$. The corresponding state space is the one shown in Figure 5.11. For simplicity, only the new algebraic expressions are put in each state. Notice that the distance of a state from the root is equal to its index. Dynamic programming starts at the root and works its way towards the leaves of the tree in stages. In stage k it computes the cost of the paths leading to states of index k. Eventually it computes the cost from the root to all the leaves and chooses the cheapest one. □

5.3.2. Implementation Issues for Dynamic Programming

Although the dynamic programming optimization algorithm has not been implemented, there are a few comments on the formulation of the algorithm in the previous section that are appropriate. These should help in a future implementation.

The state space of any real world application that is worth the effort optimizing is large. The computational requirements of applying the standard procedure of

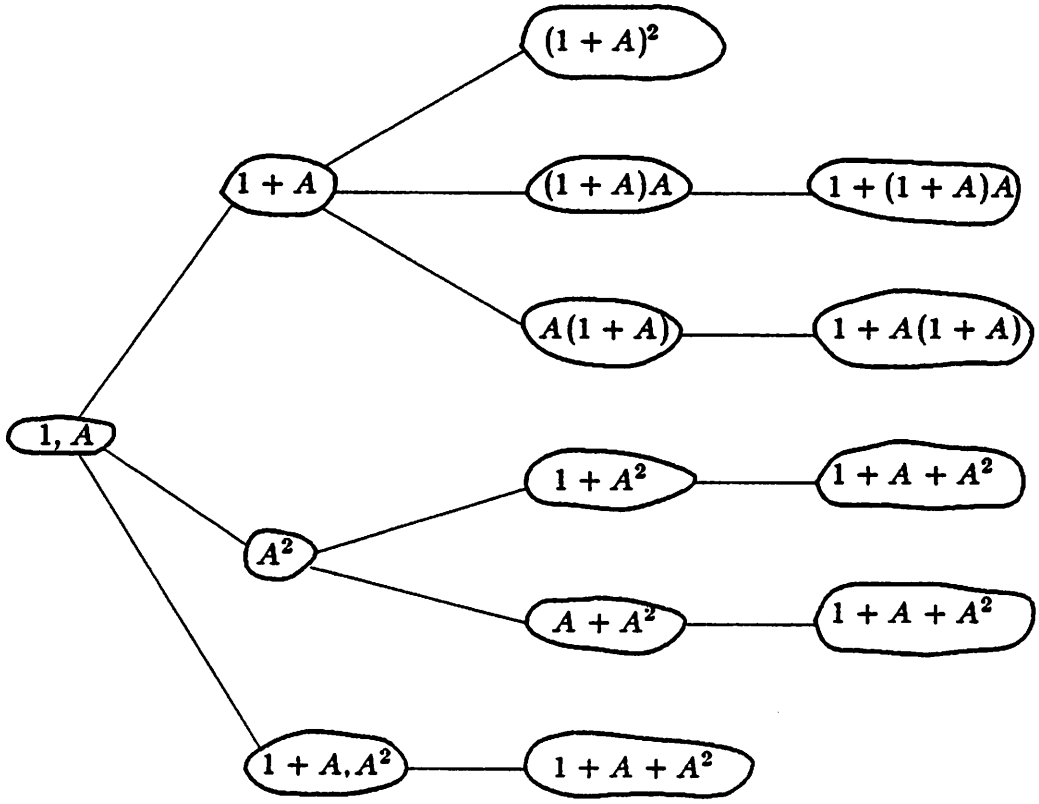

Figure 5.11. State space for the computation of $1 + A + A^2$.

dynamic programming in such a large state space may be prohibitive, let alone the space requirements of each state. Improvements can be made by changing the way states are created and visited. At every stage, dynamic programming processes all the states of some index k and expands them to get all the states of index $k+1$. Instead, *Best-First* search can be used in the state space, expanding only the state that has the minimum cost associated with the path from the root to it [Rich83]. Thus, the search is guided to move towards the most promising direction always. Likewise, heuristic search algorithms (e.g. A*) may be used for the same purpose. We believe that approaches like these are more appropriate for this environment.

There are also some other forms of heuristics that, if applied carefully, reduce the size of the state space without missing the actual optimal strategy to compute A^*. One such heuristic is to keep and expand only the M best paths (see Figure 5.12).

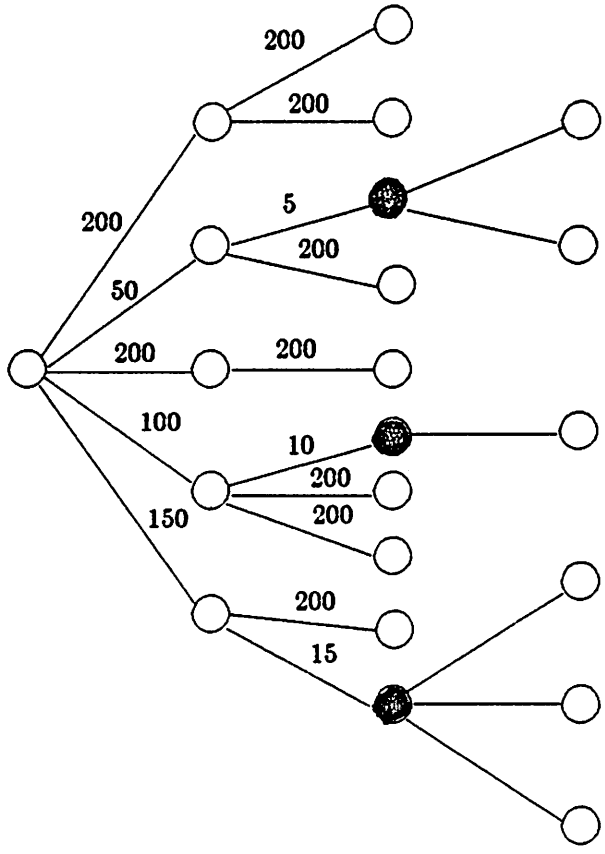

Figure 5.12. Heuristic pruning of state space with N=2 and M=3.

N and M are parameters of the heuristic algorithm, whose values must be tuned for performance and accuracy. The larger they are the slower the execution of the optimization algorithm but the higher the probability that the path found is the actual optimum.

Pruning down the state space in this way seems even more appropriate in an actual system, where the cost function is only an estimate of the actual cost. In such

an environment it may be desirable to interleave parts of the execution of the optimization algorithm (dynamic programming) with the actual execution of database operations. The following is a possible execution paradigm: Execute the optimization algorithm for N steps. Apply to the database all the operations involved in the first transition of the best path so far. Keep the best M paths rooted at the state that this transition leads and continue in the same manner. The tree is pruned down by both keeping only the subtree rooted at the end of the path that has already been executed on the database and by keeping only the best M paths in this subtree. One more advantage of this approach is that the optimization algorithm gets feedback information about the actual cost of the operations concerned.

A final comment on the structure of the state space S_D is that the cost-indifference of associativity of addition is not taken into account. This is partly true for the commutativity of addition as well (when two expressions of the same depth are added). Thus, many states are created that are known to be equivalent (no cost difference). These equivalence classes in the state space must be identified so that only one state is created for them. Each root-to-leaf path must be distinctly different from the rest. In addition, the state space S_D can be further reduced by applying results like Theorem 5.1, which has been completely ignored in the construction of Section 5.3.1.

5.4. Simulated Annealing vs. Dynamic Programming

Since we have implemented simulated annealing but not dynamic programming it is almost impossible and unfair to compare the two algorithms. However, there are

some similarities and some differences between them that should be emphasized.

A certain form of duality exists between the two algorithms. Both algorithms are applied on a state space (S_A and S_D). The two state spaces are different, yet related in a nice way. Each state in S_A is a strategy to compute A^*. The same is true for each leaf (or root-to-leaf path) in S_D. In that sense, each leaf state in S_D corresponds to a state in S_A.

On the other hand, there are many distinct differences between the two algorithms also. In simulated annealing a complete solution is manipulated at any time, whereas in dynamic programming the solutions are gradually built from simpler expressions. Because of this, the cost functions c_A and c_D are defined differently. On the one hand, c_A is defined on S_A, the state space, whereas on the other hand c_D is defined on $S_D X S_D$, i.e. the transitions from state to state.

The following can be drawn from a first qualitative comparison between simulated annealing and dynamic programming. Even though both are extremely time consuming, simulated annealing is much more so than dynamic programming. It requires a large number of steps to converge. Moreover, simulated annealing, being a probabilistic algorithm, relies on convergence properties to give a satisfactory answer, whereas dynamic programming, if no heuristics are used, guarantees to give the global minimum.

On the other hand, dynamic programming has certain disadvantages compared to simulated annealing. Its space requirements are much larger. Not only the state space is larger ($| S_D | \geq | S_A |$) but at any time it has to maintain all the states in the

expansion frontier. Simulated annealing stores only one state at a time. Also the transitions in dynamic programming are more complex, mainly because not all additions and multiplications between algebraic expressions in a state are legal (see definition of S_D and $N_D(s)$). Finally, some paths in dynamic programming lead to states that have computed not only A^* but some useless expressions as well. Such unnecessary effort does not occur in simulated annealing, since it is always some computation of A^* that is manipulated.

Which of the two approaches gives the most effective optimization algorithm remains to be seen. Complete implementations of both and extensive experimentation are needed to draw useful conclusions in this direction. However, both algorithms may serve a useful role in another direction. There may be many cases that one of the few traditional approaches to answer the query (e.g. semi-naive evaluation or smart evaluation) is in fact optimal. If such a situation arises, it is unwise to apply any optimization algorithm with a huge execution cost only to end up applying one of the well understood techniques. Simulated annealing or dynamic programming may well serve as a testbed, within which the strategy state space for various cases of recursion can be explored so that possible performance patterns are identified.

5.5. Summary

The significant cost difference between the various strategies to compute A^* justifies the development of fast and reliable optimization algorithms for recursion. Simulated annealing and dynamic programming have been used as models to design two such optimization algorithms. The approach taken by the two algorithms is con-

siderably different. Each one has its particular advantages and disadvantages. Despite the fact that the experience with the two algorithms has been extremely limited the initial results are encouraging.

CHAPTER 6

CONCLUSIONS AND FUTURE RESEARCH

Deductive database systems have the complete functionality of relational database systems and, in addition, the capability to infer new facts and rules from others that are explicitly stored in the database. Inference on small amounts of facts and rules has been the primary service provided by expert systems and logic programming systems. With the evolution of artificial intelligence, computer aided design and manufacturing, and decision support applications, the significance of inference on large knowledge bases is now realized as well.

Since the beginning of the development of deductive database systems, Horn clauses have been identified as the most important subset of first order logic, with respect to inference in database systems. Relational database systems have supported a limited subset of Horn clauses in the form of view definitions and integrity constraints. Even though there are many small differences between a view definition and a Horn clause, the single most important one is that a Horn clause can be recursive. There are significant computational challenges faced by a system supporting recursion, and this dissertation has focused on answering some of them.

6.1. Conclusions

Recursion in database systems has been traditionally studied within a first order logic framework, Horn clauses in particular. We have presented a different formula-

tion of the problem, based on algebraic structures. We have embedded a significant subset of the relational algebra operators in a closed semiring. In this setting, Horn clauses correspond to equations, and answering queries corresponds to solving these equations. Even though the techniques and tools that logic offers for Horn clauses have proved to be powerful for the study of many problems related to recursion, there are other problems that cannot be addressed as nicely. The most severe limitation stems from the absence of an explicit representation of query answers in terms of the underlying Horn clauses. The solution is described algorithmically, e.g. found by application of resolution theorem proving. The algebraic framework introduced in Chapter 2 allows the equation solutions (query answers) to be explicitly represented. In addition, the properties of a closed semiring allow the manipulation of this explicit representation of the query answer. Thus, we have been able to both address some of the problems raised (like the possible equivalence of linear and nonlinear recursion) and identify new algorithms to answer queries. We believe that algebraic manipulation of queries will be a significant part of the query optimizer. This, along with the particular semantics of the individual queries and statistics about the database at the given point of time, will be used to produce the plan for the query execution. This is opposite from the case of the regular, nonrecursive queries, where as experience has shown, algebraic manipulation of the query is of no help to the optimizer.

Uniformly bounded recursion has been studied using the Horn clause representation. We have considered a restricted class of linear recursive Horn clauses and have demonstrated that some of them are equivalent to a finite number of nonrecursive ones. Such a Horn clause has been modeled by a weighted directed graph. Uniform

boundedness has been proved to be equivalent to the property that the graph has no cycles of nonzero weight. Finally, we have indicated some possible implications of this result in the construction of efficient algorithms to process recursive statements.

The power of the operator algebra developed in Chapter 2 is found in its ability to reveal alternative algorithms to answer queries. We have identified some such algorithms to materialize recursively defined relations. We have analyzed and experimented with a few promising ones, namely the smart and the minimal algorithm, which make a trade-off between the number of multiplications performed between operators and their individual costs. For the transitive closure of trees, both the analysis and experimental results have shown that, in comparison with the traditional ones, namely the naive and the semi-naive algorithm, the new algorithms perform better for more shallow relations than for deeper ones. Nevertheless, the relation size (width/depth), at which the new algorithms start performing worse that the old ones, is large. This makes the new algorithms attractive for many of the expected cases. Finally, comparing the two new algorithms with each other shows again that, for recursive computations, minimizing the number of multiplications pays off, unless the relations are large.

In general, recursion is expected to be a major source of processing inefficiency. Query optimization for recursion is even more important than it is for nonrecursive relational queries. With the respect to query optimization, the ability to represent explicitly and manipulate the query answer is important. We have have devised two optimization algorithms, one based on simulated annealing and another based on dynamic programming. Both have been applied on state spaces that are constructed

from the equivalent algebraic forms of the transitive closure of relational operators. Simulated annealing has been implemented and experiments have been performed for a limited class of relational operators. Our initial experience is that there are many strategies that perform better than the traditional ones, thereby supporting the previous comments on the importance of query optimization for recursion.

6.2. Future Research

This dissertation has merely touched some of the issues related to the efficient support of recursion in a deductive database system. Most of them need further investigation and improvement. The following are some suggestions in this direction.

6.2.1. Algebraic Formulation

Results pertaining to all aspects of query processing and optimization for recursion appear to be limited by the fact that relational operators form a closed semiring rather than a ring or a richer algebraic structure. Perhaps we need to embed the operators in such a larger algebraic structure. In terms of processing, this may mean that more information needs to be kept at each stage of iteration (assuming that recursive queries are answered by some iterative program), but the additional information requirements is compensated by greater applicability of cost effective processing techniques.

6.2.2. Uniformly Bounded Recursion

Many issues related to uniform boundedness have not been answered yet. Necessary and sufficient conditions need to be obtained for more general classes of recursive

Horn clauses, by removing some of the restrictions R1 to R5 of Section 3.2. Recently, it has been shown that both boundedness and uniform boundedness are undecidable in the presense of multiple Horn clauses [Gaif86]. However, the question is open for the case of a single recursive Horn clause. The partial results known in this direction seem to indicate that, even if the question is decidable, an efficient decision procedure is highly unlikely. The algebraic formulation may give new insights in this direction as well.

6.2.3. Query Processing Algorithms

The scope of the analysis and experiments with the smart and minimal algorithms of Chapter 4 is quite limited. There is a number of directions that need to be investigated further. Wider range of sizes for the relations, different structure of the relations (other than trees), wider range of the buffer pool size, and other forms of operators (other than transitive closure of binary relation) are the directions to take.

In addition, all the interesting strategies need to be actually implemented (rather than simulated), so that there is no limitation imposed on the experimentation by any preexisting inefficiencies. In that respect, analyzing and testing the semi-naive, smart, and minimal algorithms using join strategies other than merge-scan is essential. Since sorting costs become prohibitive as the relations grow, hash-join techniques seem promising, as was pointed out in [Vald86]. Also, avoiding the creation and destruction of temporaries as much as possible is another area for investigation.

The smart and minimal strategies of Chapter 4 represent only a small sample of fast query processing strategies for recursion. There is a wide spectrum of potentially

cost-effective techniques, which have been barely mentioned in this dissertation. The swapping strategy of Section 4.4 is one such example. Careful analysis and experimentation with a wide variety of relational operators is needed to get an accurate measure of its effectiveness.

Finally, there are two key ideas in connection to efficient query processing that have been neglected from any study of recursion that we are aware of: (a) precomputation and (b) decomposition. The idea of precomputation is to augment the database with additional relations derived from the database, so that some recursive queries on the original database are no longer recursive on the augmented one. The main question that arises is, for a given augmentation, what the set of such queries is. And conversely, for a given class of recursive queries, what the augmentation must be. The known results on this are fragmentary. These questions are currently being investigated for views in the context of regular relational databases [Blak86] and are expected to be crucial when the "view" is defined recursively.

Decomposition is related to precomputation. The problem here is to reduce the complexity of recursion. For example, if A can be expressed in terms of B and C, both of which are simpler operators, when can A^* be expressed in terms of B^* and C^*? An example of that has been given in Section 3.8, where one of the operators is bounded. Also, the counting strategy [Banc86a], which was mentioned in Section 4.1.3, applies some form of decomposition. The effectiveness of the above suggests that decomposition may play an important role in the future, and its applicability should be heavily exploited.

6.2.4. Query Optimization Algorithms

Complete implementations of both the simulated annealing and the dynamic programming algorithms are necessary. These should take into account the problems of the current formulations of the state spaces and the algorithms mentioned at the end of Sections 5.2 and 5.3. It may well be that a large part of the state space, constructed from all the equivalent algebraic expressions of the transitive closure of a relational operator, is unnecessary, i.e. most of the states represent strategies that are never optimal. The question then is what part of the state space is the useful one. Any major result in this direction will increase the effectiveness of the optimization algorithms significantly.

BIBLIOGRAPHY

[Ackl85]
Ackley, D. H., G. E. Hinton, and T. J. Sejnowski, "A Learning Algorithm for Boltzmann Machines", *Cognitive Science* **9** (1985), pages 147-169.

[Aho74]
Aho, A., J. Hopcroft, and J. Ullman, *The Design and Analysis of Computer Algorithms,* Addison Wesley, Reading, MA, 1974.

[Aho79a]
Aho, A. and J. Ullman, "Universality of Data Retrieval Languages", in *Proceedings of the 6th ACM Symposium on Principles of Programming Languages,* San Antonio, TX, January 1979, pages 110-117.

[Aho79b]
Aho, A., Y. Sagiv, and J. Ullman, "Equivalences Among Relational Expressions", *SIAM Computing Journal* **8**, 2 (May 1979), pages 218-246.

[Arag84]
Aragon, C. R., D. S. Johnson, L. A. Megeoch, and C. Schevon, *Optimization by Simulated Annealing: An Experimental Evaluation,* unpublished manuscript, October 1984.

[Astr76]
Astrahan, M. et al., "System R: Relational Approach to Database Management", *ACM Transactions on Database Systems* **1**, 2 (June 1976), pages 97-137.

[Banc85]
Bancilhon, F., "Naive Evaluation of Recursively Defined Relations", in *Proceedings of the Islamorada Workshop on Large Scale Knowledge Base and Reasoning Systems,* Islamorada, FL, February 1985.

[Banc86b]
Bancilhon, F. and R. Ramakrishnan, "An Amateur's Introduction to Recursive Query Processing Strategies", in *Proceedings of the 1986 ACM-SIGMOD Conference on the Management of Data,* Washington, DC, May 1986, pages 16-52.

[Banc86a]
Bancilhon, F., D. Maier, Y. Sagiv, and J. D. Ullman, "Magic Sets and Other Strange Ways to Implement Logic Programs", in *Proceedings of the 5th ACM SIGMOD-SIGACT Symposium on Principles of Database Systems,* Boston, MA, March 1986, pages 1-15.

[Baye84]
Bayer, R., "Query Evaluation and Recursion in Deductive Database Systems", in *Proceedings of the Islamorada Workshop on Knowledge Base Management Systems,* Islamorada, Florida, 1984.

[Blak86]
Blakeley, J. A., P. A. Larson, and F. W. Tompa, "Efficiently Updating Materialized Views", in *Proceedings of the 1986 ACM-SIGMOD Conference on the Management of Data,* Washington, DC, May 1986, pages 61-71.

[Blas76]
Blasgen, M. W. and K. P. Eswaran, *On the Evaluation of Queries in a Relational Data Base System,* Research Report RJ-1745, IBM San Jose, April 1976.

[Bond76]
Bondy, J. A. and U. S. R. Murty, *Graph Theory with Applications,* North Holland, 1976.

[Carr79]
Carre, B., *Graphs and Networks,* Oxford University Press, Oxford, England, 1979.

[Chan76]
Chandra, A. K. and P. M. Merlin, "Optimal Implementation of Conjunctive Queries in Relational Data Bases", in *Proc. 9th Annual ACM Symposium on Theory of Computing,* Boulder, CO, May 1976, pages 77-90.

[Clif61]
Clifford, A. H and G. B. Preston, *The Algebraic Theory of Semigroups,* Americal Mathematical Society, Providence, RI, 1961.

[Clif83]
Clifford, J., M. Jarke, and Y. Vassiliou, "A Short Introduction to Expert Systems", *Database Engineering* **6**, 4 (December 1983).

[Cloc81]
Clocksin, W. F. and C. S. Mellish, *Programming in Prolog,* Springer Verlag, 1981.

[Codd70]
Codd, E. F., "A Relational Model of Data for Large Shared Data Banks", *CACM* **13**, 6 (1970), pages 377-387.

[Cosm86]
Cosmadakis, S. and P. Kanellakis, "Parallel Evaluation of Recursive Rule Queries", in *Proceedings of the 5th ACM SIGMOD-SIGACT Symposium on Principles of Database Systems,* Boston, MA, March 1986, pages 280-293.

[Dahl82]
Dahl, V., "On Database Systems Development through Logic", *ACM TODS* **7**, 1 (March 1982), pages 102-123.

[Daya84]
Dayal, U. et al., *"Knowledge-Oriented Database Management",* Technical Report, CCA-84-02, Computer Corporation of America, Cambridge, MA, 1984.

[DeWi84]
DeWitt, D. J., R. H. Katz, F. Olken, L. D. Shapiro, M. R. Stonebraker, and D. Wood, "Implementation Techniques for Main Memory Database Systems", in *Proceedings of the 1984 ACM-SIGMOD Conference,* Boston, MA, June 1984.

[Demo86]
Demo, B., *Recursive vs. Iterative Schemes for Least Fix Point Computation in Logical Databases,* unpublished manuscript, Universita di Torino, 1986.

[Deva86]
Devanbu, P. and R. Agrawal, *Some Considerations on Moving Selections into Fixpoint Queries,* unpublished manuscript, AT&T Bell Laboratories, 1986.

[Dwor84]
Dwork, S., P. C. Kanellakis, and J. C. Mitchell, "On the Sequential Nature of Unification", *Journal of Logic Programming* **1** (1984), pages 35-50.

[Ende72]
Enderton, H. B., *A Mathematical Introduction to Logic,* Academic Press, New York, N.Y., 1972.

[Feig77]
Feigenbaum, E. A., "The Art of Artificial Intelligence: Themes and Case Studies in Knowledge Engineering", in *Proc. 5th IJCAI,* 1977, pages 1014-1029.

[Gaif86]
Gaifman, H., *NAIL communication,* February 1986.

[Gall78]
Gallaire, H. and J. Minker, *Logic and Data Bases,* Plenum Press, New York, N.Y., 1978.

[Gall81a]
Gallaire, H., Impacts of Logic on Data Bases, *Proc. 7th International VLDB Conference,* Cannes, France, August 1981, pages 248-259.

[Gall81b]
Gallaire, H., J. Minker, and J. M. Nicolas, *Advances in Data Base Theory, Vol. 1,* Plenum Press, New York, N.Y., 1981.

[Gall83]
Gallaire, H., J. Minker, and J. M. Nicolas, *Advances in Data Base Theory, Vol. 2,* Plenum Press, New York, N.Y., 1983.

[Gall84]
Gallaire, H., J. Minker, and J. M. Nicolas, "Logic and Databases: A Deductive Approach", *ACM Computing Surveys* **16**, 2 (June 1984), pages 153-185.

[Gutt84]
Guttman, A., *"New Features for Relational Database Systems to Support CAD Applications"*, PhD Thesis, University of California, Berkeley, CA, June 1984.

[Haje85]
Hajek, B., *Cooling Schedules for Optimal Annealing,* unpublished manuscript, January 1985.

[Han86]
Han, J. and H. Lu, " Some Performance Results on Recursive Query Processing in Relational Database Systems", in *Proceedings International Conference on Data Engineering*, Los Angeles, CA, January 1986, pages 533-539.

[Hash79]
Hashiguchi, K., "A Decision Procedure for the Order of Regular Events", *Theoretical Computer Science* **8** (1979), pages 69-72.

[Haye85]
Hayes-Roth, F., "Rule-Based Systems", *CACM* **28**, 9 (September 1985), pages 921-932.

[Hens84]
Henschen, L. and S. Naqvi, "On Compiling Queries in Recursive First-Order Databases", *JACM* **31**, 1 (January 1984), pages 47-85.

[Ioan84]
Ioannidis, Y. E., L. D. Shinkle, and E. Wong, "Enhancing INGRES with Deductive Power" (Position Paper), in *Proceedings of the 1st International Workshop on Expert Database Systems*, Kiawah Isl., SC, October 1984, pages 847-850.

[Ioan86]
Ioannidis, Y. E. and E. Wong, "An Algebraic Approach to Recursive Inference", in *Proceedings of the 1st International Conference on Expert Database Systems*, Charleston, South Carolina, April 1986, pages 209-223.

[Jark84]
Jarke, M., J. Clifford, and Y. Vassiliou, "An Optimizing Prolog Front-End to a Relational Query System", in *Proceedings of the 1984 ACM-SIGMOD Conference on the Management of Data*, Boston, MA, June 1984.

[Kers86]
Kerschberg, L., *Expert Database Systems, Proceedings from the First International Workshop,* Benjamin/Cummings, Inc., Menlo Park, CA, 1986.

[Kife85]
Kifer, M. and E. Lozinskii, *Query Optimization in Logic Databases,* Technical report, SUNY, Stonybrook, June 1985.

[Kirk83]
Kirkpatrick, S., C. D. Gelatt, Jr., and M. P. Vecchi, "Optimization by Simulated Annealing", *Science* **220**, 4598 (May 1983), pages 671-680.

[Kowa83]
Kowalski, R. A., "Logic Programming", in *Information Processing 83*, edited by R. E. Mason, North Holland, 1983, pages 133-145.

[Kowa84]
Kowalski, R. A., "Logic as a Database Language", in *Proc. 3rd British National Conference on Databases*, edited by J. Longstaff, Leeds, U.K., July84, pages 103-132.

[Lafo85]
Lafortune, S. and E. Wong, *A State Transition Model for Distributed Query Processing*, (to appear in ACM-TODS), Memorandum No. UCB/ERL M85/75, University of California, Berkeley, September 1985.

[Lars78]
Larson, R. E. and J. L. Casti, *Principles of Dynamic Programming, Part I*, Marcel Dekker, Inc., New York, N.Y., 1978.

[Lewi81]
Lewis, H. and C. Papadimitriou, *Elements of the Theory of Computation*, Prentice Hall, Englewood Cliffs, NJ, 1981.

[Lozi85]
Lozinskii, E., "Evaluating Queries in Deductive Databases by Generating", in *Proc. 11th IJCAI*, Los Angeles, CA, 1985, pages 173-177.

[Mack86]
Mackert, L. F. and G. M. Lohman, "R* Validation and Performance Evaluation for Local Queries", in *Proceedings of the 1986 ACM-SIGMOD Conference on the Management of Data*, Washington, DC, May 1986, pages 84-95.

[Mand77]
Mandel, A. and I. Simon, "On Finite Semigroups of Matrices", *Theoretical Computer Science* **5** (1977), pages 101-111.

[Mink83]
Minker, J. and J. M. Nicolas, "On Recursive Axioms in Deductive Databases", *Information Systems* **8**, 1 (1983), pages 1-13.

[Mitr85]
Mitra, D., F. Romeo, and A. Sangiovanni-Vincentelli, "Convergence and Finite-Time Behavior of Simulated Annealing", in *Proc. 24th Conference on Decision and Control*, Ft. Lauderdale, FL, December 1985.

[Naug86]
Naughton, J., "Data Independent Recursion in Deductive Databases", in *Proceedings of the 5th ACM SIGMOD-SIGACT Symposium on Principles of Database Systems*, Boston, MA, March 1986, pages 267-279.

[Nico78]
Nicolas, J. M. and H. Gallaire, "Data Base: Theory vs. Interpretation", in *Logic and Data Bases*, edited by H. Gallaire and J. Minker, Plenum Press, New York, N.Y., 1978, pages 33-54.

[Nico83]
Nicolas, J. M. and K. Yazdanian, "An Outline of BDGEN: A Deductive DBMS", in *Information Processing 83*, edited by R. E. Mason, North Holland, 1983, pages 711-717.

[Park86]
Parker, R. et al., "Logic Programming Databases", in *Expert Database Systems, Proceedings from the First International Workshop*, edited by L. Kerschberg, Benjamin/Cummings, Inc., Menlo Park, CA, 1986, pages 35-48.

[Pele84]
Peleg, D., "A Generalized Closure and Complement Phenomenon", *Discrete Mathematics* **50** (1984), pages 285-293.

[RTI84]
RTI,, *INGRES Reference Manual, Version 2.1*, July 1984.

[Rich83]
Rich, E., *Artificial Intelligence*, McGraw-Hill, New York, N.Y., 1983.

[Robi65]
Robinson, J. A., "A Machine Oriented Logic Based on the Resolution Principle", *JACM* **12**, 1 (January 1965), pages 23-41.

[Rome84]
Romeo, F., A. Sangiovanni-Vincentelli, and C. Sechen, "Research on Simulated Annealing at Berkeley", in *Proc. 1984 IEEE International Conference on Computer Design*, Port Chester, N.Y., October 1984, pages 652-657.

[Rome85]
Romeo, F., A. Sangiovanni-Vincentelli, and ""Probabilistic Hill Climbing Algorithms: Properties and Applications", , in *Proc. 1985 Chapel Hill Conference on VLSI*, edited by H. Fuchs, Computer Science Press, Chapel Hill, N.C., 1985, pages 393-417.

[Rose86]
Rosenthal, A., S. Heiler, U. Dayal, and F. Manola, "Traversal Recursion: A Practical Approach to Supporting Recursive Applications", in *Proceedings of the 1986 ACM-SIGMOD Conference on the Management of Data*, Washington, DC, May 1986, pages 166-176.

[Sagi80]
Sagiv, Y. and M. Yannakakis, "Equivalences Among Relational Expressions with the Union and Difference Operators", *JACM* **27**, 4 (October 1980), pages 633-655.

[Sagi85]
Sagiv, Y., "On Computing Restricted Projections of Representative Instances", in *Proceedings of the 4th ACM SIGMOD-SIGACT Symposium on Principles of Database Systems*, Portland, OR, March 1985, pages 171-180.

[Scio84]
Sciore, E. and D. Warren, "Towards an Integrated Database-Prolog System", in *Proceedings of the 1st International Workshop on Expert Database Systems*, Kiawah Isl., SC, October 1984, pages 801-815.

[Sech86b]
Sechen, C. and A. Sangiovanni-Vincentelli, "TimberWolf 3.2: A New Standard Cell Placement and Global Routing Package", in *Proc. Design Automation Conference*, Las Vegas, NV, June 1986.

[Sech86a]
Sechen, C., *private communication,* June 1986.

[Seli79]
Selinger, P. et al., "Access Path Selection in a Relational Data Base System", in *Proceedings of the 1979 ACM-SIGMOD Conference on the Management of Data*, Boston, MA, June 1979, pages 23-34.

[Shap80]
Shapiro, S. and D. McKay, "Inference with Recursive Rules", in *Proc. 1st Annual National Conference on Artificial Intelligence*, Palo Alto, CA, August 1980.

[Stef82]
Stefik, M., J. Aikins, R. Balzer, J. Benoit, L. Birnbaum, F. Hayes-Roth, and E. Sacerdoti, *"The Organization of Expert Systems: A Prescriptive Tutorial"*, (to appear in ACM-TODS), Technical Report, VLSI-82-1, XEROX-PARC, Palo Alto, CA, January 1982.

[Ston75]
Stonebraker, M., "Implementation of Integrity Constraints and Views by Query Modification", in *Proceedings of the 1975 ACM-SIGMOD Conference on the Management of Data*, San Jose, CA, June 1975, pages 23-34.

[Ston76]
Stonebraker, M., E. Wong, P. Kreps, and G. Held, "The Design and Implementation of INGRES", *ACM Transactions on Database Systems* **1**, 3 (September 1976), pages 189-222.

[Tars55]
Tarski, A., "A Lattice Theoretical Fixpoint Theorem and its Applications", *Pacific Journal of Mathematics* **5** (1955), pages 285-309.

[Ullm85]
Ullman, J., "Implementation of Logical Query Languages for Databases", *ACM TODS* **10**, 3 (September 1985), pages 289-321.

[Vald86]
Valduriez, P. and H. Boral, "Evaluation of Recursive Queries Using Join Indices", in *Proceedings of the 1st International Conference on Expert Database Systems*, Charleston, SC, April 1986, pages 197-208.

[VanE76]
VanEmden, M. H. and R. A. Kowalski, "The Semantics of Predicate Logic as a Programming Language", *JACM* **23**, 4 (January 1976), pages 733-742.

[Viei86]
Vieille, L., "Recursive Axioms in Deductive Databases: The Query / Subquery Approach", in *Proceedings of the 1st International Conference on Expert Database Systems*, Charleston, SC, April 1986, pages 179-193.

[Wile83]
Wilensky, R., *LISPcraft*, 1983.

[Wong76]
Wong, E. and K. Youssefi, "Decomposition - A Strategy for Query Processing", *ACM Transactions on Database Systems* **1**, 3 (September 1976), pages 223-241.

[Zani85]
Zaniolo, C., "The Representation and Deductive Retrieval of Complex Objects", *Proc. 11th International VLDB Conference*, Stockholm, Sweden, August 1985, pages 458-469.