Graph–Based Processing of Logic Programs

Dietmar Seipel
University of Tübingen
Sand 13, D – 72076 Tübingen, Germany
seipel@informatik.uni-tuebingen.de

Abstract

A lot of practical recursive logic queries to a deductive database are expressed by linear recursive DATALOG programs with exactly one recursive rule, so-called **linear sirups**. We give a characterization of k-sided linear DATALOG sirups – the concept was introduced by Naughton in [22] – based on a graph model and present a normal form to which every linear DATALOG sirup can be transformed. Normalized linear DATALOG sirups can be **decomposed** by a counting technique into k lower-dimensional linear sirups which can be evaluated in parallel by very efficient query evaluation algorithms. For certain **sets** of linear sirups defining the same intensional predicate we can also derive a decomposition approach from the graph model.

Another technique based on the prefixes of compiled expansions transforms a set of chain rules with binary predicate symbols, a so-called **CFG program**, into a **transition graph**. Common subexpressions between different compiled formulas can be detected by the graph minimization algorithm of Huffman and Moore, cf. [16]. The minimized graph is used to guide the ordered, complete, but non-redundant evaluation of the program. The resulting interformula optimization is a generalization of the well-known wavefront techniques, which use intermediate results from one iteration in following iterations.

Keywords: deductive databases, query optimization, linear recursion, chain rules, wavefront techniques, counting techniques

1 Introduction

Deductive databases use **predicate calculus** as a powerful, declarative query language to retrieve data from relational databases. A query is formulated as a logic program – more precisely a set of definite horn clauses – and a query goal given by a single atom. The possibly recursive program is evaluated iteratively in a **set–oriented** fashion based on the primitive operations of relational algebra – selection, projection, join – which can be executed efficiently on databases.

Wavefront techniques, cf. Han and Lu [13], consider the complete binding–passing behaviour of a logic program. They deal with the **expansion** of a program LP w.r.t. a query goal G, which is given by an (infinite) sequence of conjunctions of atoms – each conjunction called a string – derived via repeated SLD–resolution beginning with goal G. The strings are partitioned into special subsets of (transitively) connected atoms which are evaluated separately. Subsequently the relations for the intensional predicates in LP are composed.

We derive optimization concepts for programs with one recursive rule by investigating the question how to partition the strings into so–called waves such that

(i) the result of a wave in one string is useful for deriving the results of related waves in other strings (multiple query optimization),

(ii) the intermediate wave–results form a suitable decomposition of the final result.

For a **linear recursive** rule we define a graph–representation, the augmented substitution graph, which reflects the variable sharing within the rule. There is one node for each argument position of the intensional atoms and one node for each extensional atom. The edges connect those nodes that share variables. We show how this graph describes the growth of the waves and how the rule can be decomposed into a finite number of certain less complex linear recursive rules. This generalizes the **counting technique**, cf. [2], from linear DATALOG sirups in linear variable pattern to arbitrary linear DATALOG sirups. Note, that the general counting techniques of [26], [27] do not identify the waves – their emphasis is on the propagation of query bindings. For a set of linear recursive rules defining the same intensional predicate and having only unary and binary extensional predicate symbols we can also derive a decomposition approach from the set of their augmented substitution graphs.

There are several approaches for processing recursive **CFG programs**, e.g. the multi–formula merged processing of Han and Henschen [12]. This stack–directed processing algorithm for bilinear CFG sirups generates many similar strings, whose processing can be optimized by sharing and merging of the formulas. CFG programs are related to context–free languages. Some expressiveness results for them are given in [8].

The approach for general CFG programs presented here is based on ideas from automata theory and formal languages. The strings are processed by an algorithm operating on their corresponding **transition graph**, which has a node for each string–prefix and an edge from each node to its direct suffixes. Thus, several types of redundancy in the processing of the strings can be avoided. There is a similar rewriting technique of Houtsma and Apers, cf. [17], which is also graph–based, but is only applicable for regular mutually recursive logic programs.

1.1 Basic Definitions and Notations

Syntactically, a **deductive database** $DDB = (DB, LP)$ is given by a set DB of facts, which resemble a relational database, and a logic program LP, which consists of a set of rules for deriving new facts from DB, cf. [21]. A **fact** is an atom $p(t_1, \ldots, t_n)$, where t_i is a term over variables, constants and function symbols. A goal is given by a sequence of atoms. A **rule** r is given by

$$r: \ \ p_0(X_0) \ :- \ p_1(X_1), \ldots, p_n(X_n).$$

where the atom $p_0(X_0)$ is called its head and the goal $p_1(X_1), \ldots, p_n(X_n)$ where $n \in \mathbb{N}_0$ is called its body. r is a DATALOG rule, iff it contains no function symbols in the terms of its atoms.

The **semantics** of a deductive database $DDB = (DB, LP)$ is defined according to the bottom-up or naive evaluation algorithms, cf. [3]. Notions of **equivalence** for logic programs are introduced in [25].

(i) A predicate symbol p is **intensional**, iff it occurs in the head of a rule $r \in LP$; otherwise it is **extensional**. A logic database DB is extensional, iff it contains only facts with extensional predicate symbols. The p-**database** of DB contains all facts $p(t_1, \ldots, t_n) \in DB$.

(ii) The database containing all facts $p(t_1, \ldots, t_n)$ that can be derived from DB by repeated applications of rules $r \in LP$ is denoted by $LP^\omega(DB)$.

(iii) Logic programs LP_1 and LP_2 are called p-**equivalent** w.r.t a predicate symbol p, iff the p-databases of $LP_1^\omega(EDB)$ and $LP_2^\omega(EDB)$ are equal for all extensional databases EDB.

The complexity of evaluating recursive queries mainly depends on the structure of the **recursive rules**, cf. [3], [31]. A certain subclass of linear DATALOG sirups, which we will call standard linear form sirups, is frequently considered in literature.

(iv) A logic program LP is called a **linear single recursive rule program** (linear sirup), iff it contains exactly one linear recursive rule r and some non-recursive rules, called exit rules, where

$$r: \quad p(X) \ :- \ p(U), a_1(U_1), \ldots, a_n(U_n).$$

The **dimension** of LP is the arity n' of p. The **width** of LP is n.
LP is called a **linear multiple recursive rule program** (linear murup), iff it contains only linear recursive rules defining the same predicate symbol p and some non-recursive exit rules.

(v) A linear DATALOG sirup LP is in **standard linear form (SLF)**, iff

(SLF-1) r is range–restricted, i.e. all variables in $p(X)$ occur in at least one body atom.

(SLF-2) different body atoms $a_i(U_i)$ share no common arguments.

(SLF-3) $p(X)$ contains no constants and no repeated variables.

The above assumptions (SLF-i) are made for the following reasons. Firstly, the property **(SLF-1)** guarantees the safety of the DATALOG sirup. Secondly, a sirup LP can easily be transformed to an equivalent sirup LP' having the property **(SLF-2)** by collapsing the connected components of extensional atoms in r. Thirdly, the **sidedness** of sirups is defined in terms of a special **expansion** for linear DATALOG sirups in [22], which requires the assumption **(SLF-3)**. According to Jagadish et al. [19], every linear DATALOG sirup can be simplified to an equivalent linear DATALOG sirup having this property.

Many practical examples were in an even more special normal form corresponding to context-free grammars, for which more specialized evaluation algorithms will be introduced in section 3.

(vi) A logic program LP is called a **CFG program** (context-free grammar program), iff it contains only rules r in linear variable pattern:

$$r: \quad p_0(x_1, x_{n+1}) \ :- \ p_1(x_1, x_2), \ldots, p_i(x_i, x_{i+1}), \ldots, p_n(x_n, x_{n+1}).$$

where all the occurring arguments x_i are variables.

The quality of an algorithm for **recursive query processing** depends on CPU–performance, I/O and storage costs, which are influenced by the set of relevant facts, the dimension of the logic programs, and the amount of duplication of work, since the same intensional fact may be repeatedly derived and the same extensional or intensional fact may be unnecessarily often fetched from secondary storage.

1.2 The Counting Technique for Linear Sirups

The standard **semi-naive** evaluation of a linear sirup LP iteratively increases an intermediate p-database, where all a_i-databases are relevant for deriving new facts in each iteration. In comparison to that, [19] considers so-called CFTC sirups and suggests to transform a CFTC sirup LP to an equivalent set of independently evaluable linear sirups, each depending only on one a_i-database. Unfortunately, there are CFTC sirups which are not decomposable in this way, e.g. $LP = \{r, e\}$:

$$\begin{array}{lll} r: & p(x_1, x_2, x_3) & :- \quad p(u_1, u_2, u_3), a_1(x_1, u_1, x_2), a_2(x_3, u_3, u_2). \\ e: & p(x_1, x_2, x_3) & :- \quad a_0(x_1, x_2, x_3). \end{array}$$

But the **counting method** of [2] shows that the intended decomposition is possible for all n-dimensional linear DATALOG sirups $LP = \{r, e\}$ in **linear variable pattern**:

$$\begin{array}{lll} r: & p(x_1, \ldots, x_n) & :- \quad p(u_1, \ldots, u_n), a_1(x_1, u_1), \ldots, a_n(x_n, u_n). \\ e: & p(x_1, \ldots, x_n) & :- \quad a_0(x_1, \ldots, x_n). \end{array}$$

The decomposition can be derived from the compiled exit rules e^m ($m \in \mathbb{N}_0$), which form an equivalent, infinite, non-recursive logic program $LP^\omega = \{e^m | m \in \mathbb{N}_0\}$, where u_i^j ($1 \leq i \leq n$, $j \in \mathbb{N}_+$) are different variable symbols:

$$\begin{array}{lll} e^m: & p(x_1, \ldots, x_n) & :- \quad a_1(x_1, u_1^1), a_1(u_1^1, u_1^2), \ldots, a_1(u_1^{m-1}, u_1^m), \ldots, \\ & & \qquad a_n(x_n, u_n^1), a_n(u_n^1, u_n^2), \ldots, a_n(u_n^{m-1}, u_n^m), a_0(u_1^m, \ldots, u_n^m). \end{array}$$

Consider now the 3-dimensional linear sirups $LP_i = \{r_i, e_i\}$ ($1 \leq i \leq n$) and the logic program $LP_{n+1} = \{r_{n+1}, e\}$:

$$\begin{array}{lll} r_i: & p_i(x_i, u_i^2, m+1) & :- \quad p_i(x_i, u_i^1, m), a_i(u_i^1, u_i^2). \\ e_i: & p_i(x_i, u_i^1, 1) & :- \quad a_i(x_i, u_i^1). \\ r_{n+1}: & p(x_1, \ldots, x_n) & :- \quad a_0(u_1^1, \ldots, u_n^1), p_1(x_1, u_1^1, m) \ldots, p_n(x_n, u_n^1, m). \end{array}$$

The logic program $LP' = LP_1 \cup \ldots \cup LP_{n+1}$ is p-equivalent to LP. More precisely, the sirups LP_i ($1 \leq i \leq n$) can be evaluated independently, e.g. in parallel, on the a_i-databases DB_i. LP_{n+1} combines the databases $LP_i^\omega(DB_i)$ yielding $LP^\omega(DB)$.

One desirable property of a logic program is **safety**, which means that the bottom-up evaluation terminates with a finite database for all finite extensional databases. LP' is not safe, since there exist so-called cyclic a_i-databases DB_i, where $LP_i^\omega(DB_i)$ is infinite. For acyclic databases, the naive evaluation of LP' will always terminate and efficiently compute the desired result.
The improved method of Haddad and Naughton [14] for $n = 2$ is the best among all known methods which can handle cyclic databases DB_i as well. It has the same complexity $O(h \cdot f)$ as the counting methods, where h is the number of different constant symbols in $LP \cup DB$ and f is the number of facts in DB.

2 Graph–Based Analysis of Linear Recursive Rule Programs

In the following we consider SLF sirups, since every range–restricted linear DATALOG sirup can easily be transformed to an equivalent SLF sirup.

2.1 Expansions and AS–Graphs for Linear Sirups

The **expansion** of an SLF sirup LP w.r.t. the goal $G_0 = p(X)$ is a unique sequence $(G_i \mid i \in \mathbb{N}_0)$ of SLD–resolvents, cf. Lloyd [21], defined in [22]. The resolvent G_{i+1} is obtained by resolving – via their p–atoms – G_i and a variable disjoint version of r, where each variable symbol is superscribed by $i+1$. The corresponding most general unifier is denoted by θ_{i+1}. The SLD–resolvents G_i are called strings and are considered as multisets, i.e. atoms appearing twice or more often will be treated as different elements. Strings will be denoted as follows:

$$G_j = \underbrace{p(U)\theta_j}_{=p(U^j)}, \underbrace{a_1(U_1)\theta_1, \ldots, a_1(U_1)\theta_j, \ldots, a_n(U_n)\theta_1, \ldots, a_n(U_n)\theta_j}_{reduced\_string(j)}$$

Example 2.1 *Consider the SLF sirup $LP = \{r, e\}$:*

$$\begin{array}{rlcl} r: & p(x_1, x_2, x_3, x_4, x_5, x_6) & :- & p(u_1, u_2, u_3, x_3, x_5, u_6), \\ & & & a_1(x_1, u_1), a_2(x_2, u_3), a_3(x_4, u_2), a_4(x_6). \\ e: & p(x_1, x_2, x_3, x_4, x_5, x_6) & :- & a_0(x_1, x_2, x_3, x_4, x_5, x_6). \end{array}$$

Then, the first three SLD–resolvents are given by

$$\begin{array}{lcll} G_0 & = & p(x_1, x_2, x_3, x_4, x_5, x_6) & \\ G_1 & = & p(u_1^1, u_2^1, u_3^1, x_3, x_5, u_6^1), & a_1(x_1, u_1^1), a_2(x_2, u_3^1), a_3(x_4, u_2^1), a_4(x_6) \\ G_2 & = & p(u_1^2, u_2^2, u_3^2, u_3^1, x_5, u_6^2), & a_1(x_1, u_1^1), a_2(x_2, u_3^1), a_3(x_4, u_2^1), a_4(x_6), \\ & & & a_1(u_1^1, u_1^2), a_2(u_2^1, u_3^2), a_3(x_3, u_2^2), a_4(u_6^1). \end{array}$$

Definition 2.2 (Iterates) *Consider an SLF sirup $LP = \{r, e_1, \ldots, e_h\}$ given by*

$$\begin{array}{rlcl} r: & p(X) & :- & p(U), a_1(U_1), \ldots, a_n(U_n). \\ e_i: & p(X) & :- & b_i(X). \end{array}$$

The m-th iterate of r is r^m ($m \in \mathbb{N}_+$), the compiled exit rules are e_i^m $(1 \leq i \leq h,\ m \in \mathbb{N}_0)$, the m-th iterate of LP is the linear sirup $LP^m = \{r^m, e_i^j | 1 \leq i \leq h, 0 \leq j \leq m-1\}$:

$$\begin{array}{rlcl} r^m: & p(X) & :- & p(U^m), reduced\_string(m). \\ e_i^m: & p(X) & :- & b_i(U^m), reduced\_string(m). \end{array}$$

An SLF sirup LP is p–equivalent to each of its iterates LP^m and to its infinite, non–recursive compiled version $LP^\omega = \{e_i^m | 1 \leq i \leq h, m \in \mathbb{N}_0\}$.

The concept of k-sided linear sirups introduced by Naughton [22] refers to the sharing of arguments between the different atoms in the sets $reduced\_string(m)$, i.e. to the connected components within the hypergraph that has one hyperedge $\{v_1, \ldots, v_h\}$ for each atom $a(v_1, \ldots, v_h)$ in $reduced\_string(m)$.

Example 2.3 *The iterates of the recursive rule r of the 2–dimensional linear sirup in linear variable pattern*

$$\begin{array}{llll} r: & p(x_1, x_2) & :- & up(x_1, u_1), p(u_1, u_2), down(u_2, x_2). \\ e: & p(x_1, x_2) & :- & flat(x_1, x_2). \end{array}$$

are given by

$$\begin{array}{llll} r^1: & p(x_1, x_2) & :- & up(x_1, u_1^1), p(u_1^1, u_2^1), down(u_2^1, x_2). \\ r^2: & p(x_1, x_2) & :- & up(x_1, u_1^1), up(u_1^1, u_1^2), p(u_1^2, u_2^2), down(u_2^2, u_2^1), down(u_2^1, x_2). \end{array}$$

The corresponding hypergraphs are given in figure 1. A hyperedge for the intensional p–atom in r^m is included in the picture to indicate how the hypergraph H_{m+1} is derived from H_m by hyperedge replacement.

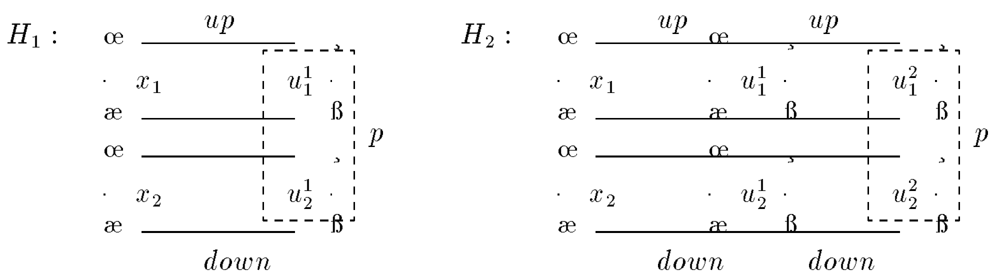

Figure 1: Hypergraph representation of rule bodies

Definition 2.4 (**k-sidedness**) *A linear* DATALOG *sirup with the property (SLF-3) is* **k-sided** *($k \in I\!N_0$), iff there exists some $c \in I\!N$, such that for any $c' > c$ some $reduced\_string(m)$ has k connected components C of size $|C| \geq c'$, and no $reduced\_string(m)$ has more than k connected components C of size $|C| \geq c'$.*

The compiled version LP^{ω}, i.e. the compiled exit rules e_i^m, can be evaluated by a **wavefront technique**, where independent frontier relations are assigned to the connected components of $reduced\_string(m)$. The frontier relations obtained for e_i^m can be used for the evaluation of $e_i^{m'}$ ($m' \geq m$), since the strings and the connected components are growing with m. The sidedness of a sirup is the number of infinitely growing connected components. E.g. the n-dimensional linear DATALOG sirup LP in linear variable pattern of section 1.2 is n-sided, and the frontier relations can be identified easily, cf. the previous example 2.3. But there is no simple rule for dealing with the frontier relations, if LP is an arbitrary SLF sirup in no special normal form.

Example 2.5 *The iterate r^3 of r in example 2.1 is given by*

$$r^3: \ p(x_1, x_2, x_3, x_4, x_5, x_6) \ :- \ p(u_1^3, u_2^3, u_3^3, u_3^2, x_5, u_6^3), reduced\_string(3).$$

where $reduced\_string(3)$ consists of the following 7 connected components:

$C_1 = \{a_1(x_1, u_1^1), a_1(u_1^1, u_1^2), a_1(u_1^2, u_1^3)\}$,
$C_2 = \{a_2(x_2, u_3^1), a_3(u_3^1, u_2^3)\}$, $C_3 = \{a_3(x_3, u_2^2), a_2(u_2^2, u_3^3)\}$, $C_4 = \{a_3(x_4, u_2^1), a_2(u_2^1, u_3^2)\}$,
$C_{5,1} = \{a_4(x_6)\}$, $C_{5,2} = \{a_4(u_6^1)\}$, $C_{5,3} = \{a_4(u_6^2)\}$.

As the basic concept for the **analysis of the expansion** of an SLF sirup we will use directed, binary, edge–labelled graphs (with parallel edges and self-loops). The substitution graphs introduced by Jagadish et al. [19] partially describe the sharing of arguments between the intensional atoms $p(U^j)$, cf. [28]. We will introduce an extended substitution graph that contains some additional nodes η_k and corresponding edges modeling this sharing of arguments completely. So far the extensional atoms are not considered in the graph model. For that purpose we will define the augmented substitution graph, which models the interaction of the intensional atoms $p(U^j)$ and the extensional atoms $a_i(U_i)\theta_j$.

Definition 2.6 (AS–graphs) *Let* $<1, m> := \{1, 2, \ldots, m\}$, *for all* $m \in I\!N_0$. $(v \xrightarrow{l} v')$ *denotes an edge* e *from node* v *to* v' *with the labeling* $l(e) \in I\!R$, *the length of* e.

(i) *The* **substitution graph (S–graph)** $G_0 = (V_0, E_0)$ *is given by:*
- $V_0 := <1, n'> \cup \{ \eta_k \mid |\eta_k| \geq 2, 1 \leq k \leq m \}$,
- $E_0 := \{ (i \xrightarrow{1} j) \mid x_i = u_j, 1 \leq i, j \leq n' \} \cup \{ (\eta_k \xrightarrow{1} i) \mid |\eta_k| \geq 2, 1 \leq k \leq m \}$,

where the sets $\eta_1, \ldots \eta_m$ *form a disjoint partition of* $<1, n'>$, *such that for all* $1 \leq i, j \leq n'$: $u_i = u_j$, *iff* $\{i, j\} \subseteq \eta_k$ *for some* $1 \leq k \leq m$.

(ii) *The* **augmented substitution graph (AS–graph)** $G_1 = (V_1, E_1)$ *is given by:*
- $V_1 := V_0 \cup \{ a_1, \ldots, a_n \}$,
- $E_1 := E_0 \cup \{ (a_i \xrightarrow{1/2} j) \mid u_j \textit{ occurs in } U_i \} \cup \{ (j \xrightarrow{1/2} a_i) \mid x_j \textit{ occurs in } U_i \}$.

Example 2.7 *The AS–graph for example 2.1 contains one node for each argument position of the (intensional) p-atoms and one node for each (extensional)* a_i*-atom. The edges connect those nodes that share variables. Firstly, an edge* (i, a_j) *labelled by* $\frac{1}{2}$ *states that the the i-th position*

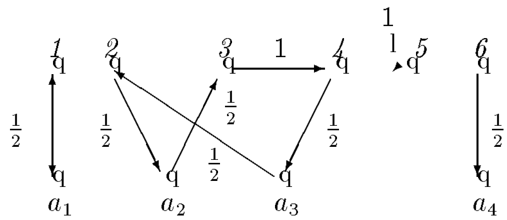

Figure 2: AS-graph of example 2.1

(x_i *) of the intensional head atom* $p(X)$ *and some position in the extensional* a_j*-atom are equal, cf. the edge* $(2, a_2)$. *Secondly, an edge* (a_j, i) *labelled by* $\frac{1}{2}$ *states that the the i-th position (* u_i *) of the intensional body atom* $p(U)$ *and some position in the extensional* a_j*-atom are equal, cf. the edge* $(a_3, 2)$. *Thirdly, an edge* (i, j) *labelled by* 1 *states that the the i-th position (* x_i *) of the intensional head atom* $p(X)$ *and the j-th position (* u_j *) of the intensional body atom* $p(U)$ *are equal, cf. the edge* $(3, 4)$. *In this example there is no node* η_k *and no edge* $(\eta_k \xrightarrow{1} i)$.

By considering the lengths (i.e. the sum of the edge labelings, where labelings of edges traversed in backward direction are counted negative) of special paths in the AS–graph, it can be decided

- which intensional atoms $p(U)\theta_j$ share arguments and
- which extensional atoms $a_i(U_i)\theta_j$ are connected within $reduced\_string(m)$.

Two corresponding lemmas concerning the S–graph and the AS–graph, respectively, are given in appendix A.

Definition 2.8 (Sidedness, Period)

(i) *A predicate symbol a_i is* **recurrent**, *iff there exists a cycle π containing a_i in the AS–graph, with the length $l(\pi) \neq 0$; it is* **isolated**, *otherwise. Let R contain the recurrent predicate symbols a_i, and let R_0 contain the isolated ones.*

(ii) *The period $m_{i,i}$ of a recurrent a_i is given by the greatest common divisor of the lengths $l(\pi)$ of all cycles π containing a_i in the AS–graph, where $l(\pi) > 0$.*

(iii) *The set R is partitioned into disjoint subsets $R_1, \ldots, R_k$, called recurrent classes, of equivalent recurrent predicate symbols: a_i and a_j are called equivalent, iff $i = j$ or there is a path from a_i to a_j in the AS–graph. R_0 is called the isolated class.*

(iv) *The period l_ν of R_ν ($1 \leq \nu \leq k$) is the (common) period of all $a_i \in R_\nu$. The characteristics of LP are its* **sidedness** $\kappa := \sum_{\nu=1}^{k} l_\nu$ *and its* **period** $\lambda := lcm\{ \ l_\nu \mid 1 \leq \nu \leq k \ \}$.

The periods of the recurrent classes are well–defined, since predicate symbols that are connected in the AS–graph have the same period, cf. [28]. The next theorem shows, that the definition of the sidedness κ above matches the definition of sidedness in [22]. A sketch of the proof is given in the appendix.

Theorem 2.9 (Sidedness of SLF sirups) *An SLF sirup LP is κ-sided. Each of its iterates LP^m ($m \in \mathbb{N}_+$) is κ-sided, too. It holds $0 \leq \kappa \leq n'$, where n' is the dimension of LP.*

Example 2.10 *The sirup LP of example 2.1 is 4–sided with the recurrent classes $R_1 = \{a_1\}$, $R_2 = \{a_2, a_3\}$, the isolated class $R_0 = \{a_4\}$, and the periods $l_1 = 1$, $l_2 = 3$. I.e. the characteristics of LP are $\kappa = 4$, $\lambda = 3$.*

2.2 Determination of the Sidedness of Linear Sirups

We will present a quadratic time–algorithm, which determines the characteristics κ and λ of an SLF sirup LP from its AS–graph.

Given a binary, edge-weighted graph $G = (V, E)$. A pair $b = (e_1, e_2)$ of edges $e_i = (v \xrightarrow{l_i} v_i)$ starting at the same node v is called a **branch**, iff $0 < l_1 \leq l_2$. Note, that the restriction $l_1 \leq l_2$ just says, that the 'shorter' edge is the first component of the branch. The **folding operator** transforms G w.r.t. b to a new graph $fold(\ G, b \ ) \ := \ (V', E')$.

(i) If $l_1 = l_2$, then a new node denoted by the string v_1v_2 is created, which replaces the old nodes v_1 and v_2, and all the old edges starting at v_1 or v_2, respectively, are replaced by new edges starting from the new node v_1v_2:

- $V' = (\ V \setminus \{v_1, v_2\}\ ) \cup \{v_1v_2\}$,
- $E' = (\ E \setminus \{\ (v_1 \xrightarrow{l} v'), (v_2 \xrightarrow{l} v') \in E \mid l \in \mathbb{R}, v' \in V\ \}\ ) \ \cup$
  $\{\ (v_1v_2 \xrightarrow{l} v') \mid (v_1 \xrightarrow{l} v') \in E \ or \ (v_2 \xrightarrow{l} v') \in E, \ l \in \mathbb{R}, v' \in V\ \}$.

(ii) If $l_1 \neq l_2$, then $V' = V$ and $E' = (\ E \setminus \{\ (v \xrightarrow{l_2} v_2)\ \}\ ) \cup \{\ (v_1 \xrightarrow{l_2 - l_1} v_2)\ \}$, i.e. the 'longer' one e_2 of the branch edges is replaced by a new edge from v_1 to v_2.

The intention of the folding operator is to simplify the structure of a graph while preserving the path lengths. After any sequence of successive applications of the folding operator to an original graph G, the set of nodes of the resulting graph G' is given by strings $v_{i_1} v_{i_2} \ldots v_{i_k}$ of original nodes. For each original node v in G there is a unique new node – denoted by v' – in G' containing the substring v. A path π of the type $(u \xrightarrow{l} w)$ in G corresponds to a path π' of the type $(u' \xrightarrow{l} w')$ in G'.

The folding algorithm FOLD below transforms a graph $G = (V, E)$ to a folded graph $G' = (V', E')$ by a sequence of successive foldings until all branches are eliminated.

Algorithm FOLD(G, G');
$G' := G$;
while (there is a branch b in G') **do**
  $G' \ := \ fold(\ G', b\ )$;

Example 2.11 *The graph G is folded to G'.*

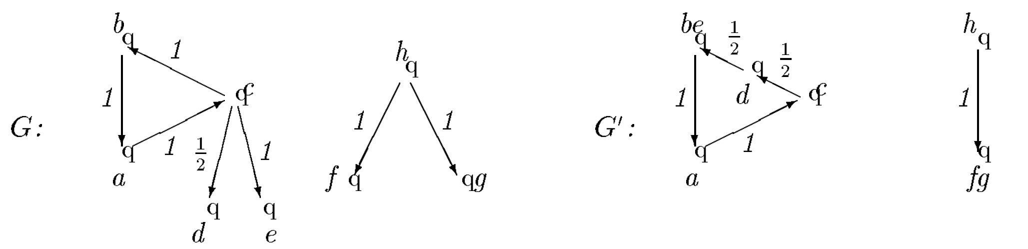

Figure 3: Folded AS–graph G' of G

This algorithm terminates for all finite graphs. Since in each folding step either the cardinality of the node set decreases while the sum of the path lengths remains invariant, or the node set remains invariant while the sum of the path lengths decreases, there will finally be no more branches.

When applied to an AS–graph $G_1 = (V_1, E_1)$ this original graph and all the intermediate graphs $G' = (V', E')$ have edge lengths $l \in \{1/2, 1\}$. We define the **potential** of a graph G' by

$$pot(G') \ = \ |V'| \ + \ |\{(v \xrightarrow{1} v') \in E'\}|,$$

which has the property that $pot(fold(G', b)) < pot(G')$ and $pot(G') \in I\!N_0$. Thus, there are at most $pot(G_1)$ successive foldings applicable to the AS–graph G_1. For a given n'–dimensional sirup LP with width n it holds $|V_1| \leq n + 1.5n'$, $|E_0| \leq 2n'$ and thus, there are at most

$$pot(V_1, E_1) \ = \ |V_1| + |E_0| \ \leq \ n + 3.5n'$$

folding operations, where each can be performed in $n + n'$ elementary operations.

Theorem 2.12 *Consider an SLF-sirup LP and its folded AS-graph* $G'_1 = (V'_1, E'_1)$.

(i) *Function* FOLD *has the Church-Rosser property (G'_1 is independent of the sequence of foldings, except for different orders of the symbols v_{i_j} in the strings $v_{i_1}v_{i_2}\dots v_{i_k}$ denoting the nodes), and its complexity is in $O(max\{n^2, n'^2\})$.*

(ii) *All paths and cycles in G'_1 are uni–directional; their nodes are pairwise not connected by a path. For each original node $v \in V_1$ there is a unique new node $v' \in V'_1$ containing the substring v.*

(iii) *Let $\pi_1, \dots, \pi_k$ be those cycles in G'_1 that traverse nodes $a'_i \in V'_1$ corresponding to the extensional predicate symbols a_i of LP. Then $R_\nu = \{a_i | a'_i \text{ is traversed by } \pi_\nu, 1 \le i \le n\}$ ($1 \le \nu \le k$) are the recurrent classes of LP whose periods l_ν are the lengths of π_ν.*

The cycles in the folded AS–graph can be determined easily, since different cycles are guaranteed to reside in different connected components of the symmetrisized graph.

Example 2.13 *There are no branches in the AS–graph of example 2.1. Thus, it equals its folded AS–graph. There exist two cycles π_1 and π_2. π_1 is traversing the recurrent class $R_1 = \{a_1\}$, π_2 traverses $R_2 = \{a_2, a_3\}$. The periods are $l(\pi_1) = l_1 = 1$ and $l(\pi_2) = l_2 = 3$.*

2.3 Normalization and Decomposition of Linear Sirups

The decomposition of SLF sirups in linear variable pattern by the counting method was based on the fact that there was one infinitely growing connected component for each recurrent a_i-atom ($1 \le i \le n$). This also holds for the more general class of KNF sirups.

Definition 2.14 (KNF) *An SLF sirup LP is in* κ**-sided normal form (KNF)***, iff for its recursive rule*

$$r: \quad p(X) \ :- \ p(U), a_1(U_1), \dots, a_n(U_n).$$

there exists a partition which splits the argument vectors X and U into $\kappa \cdot 3 + 2$ subvectors:

$$\begin{aligned} X &= (X_1, Y'_1, Z'_1, \dots, X_\kappa, Y'_\kappa, Z'_\kappa, X_S, W) \\ U &= (X_1, Y_1, Z_1, \dots, X_\kappa, Y_\kappa, Z_\kappa, U_S, W) \end{aligned}$$

Subvectors with different names are disjoint.

(i) *The i-th triples (X_i, Y'_i, Z'_i) in X and (X_i, Y_i, Z_i) in U are related by the corresponding atom $a_i(U_i)$ ($1 \le i \le \kappa$), where $U_i = (X_i, Y'_i, Y_i, Z'_i)$ and the subvector (X_i, Y'_i, Y_i) is not empty. These atoms $a_i(U_i)$ are called recurrent.*

(ii) *The argument vectors X_S and U_S respectively are connected only to atoms $a_i(U_i)$ ($\kappa+1 \le i \le n$), such that a_i does not belong to a cycle of positive length in the AS-graph. a_i is called isolated.*

(iii) *The argument vector W is not connected to any extensional atom.*

The recursive nature of a KNF sirup stems from the so-called **transitive** argument positions Y_i. E.g. in the linear DATALOG sirup in linear variable pattern of section 1.2 all positions are transitive.

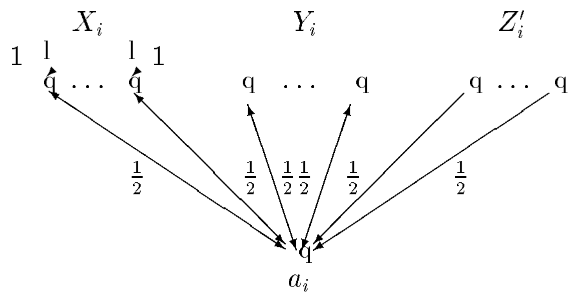

Figure 4: AS-subgraph with argument positions connected to a recurrent a_i

Every SLF sirup LP can be normalized to an equivalent KNF sirup LP' with the same sidedness and dimension. LP' is derived from the iterate LP^λ of LP, where λ is the period of LP.

Theorem 2.15 (Normalization of SLF sirups) *For every κ-sided, n'-dimensional SLF sirup LP with the recursive predicate symbol p there is a κ-sided, n'-dimensional KNF sirup LP' that is p-equivalent to LP.*

Example 2.16 *The KNF sirup $LP' = \{r', e_1, \dots, e_5\}$ for LP from example 2.1 is derived from the connected components in r^3, since $\lambda = 3$. It is 4–sided like LP and has the periods $l_i = 1$ $(1 \leq i \leq 4)$. The argument positions 1 to 4 are transitive.*

$$\begin{aligned} r' : \ p(x_1, x_2, x_3, x_4, x_5, x_6) \ &:- \ p(u_1^3, u_2^3, u_3^3, u_3^2, x_5, u_6^3), \\ &\quad\ b_1(x_1, u_1^3), b_2(x_2, u_2^3), b_3(x_3, u_3^3), b_4(x_4, u_3^2), b_5(x_6). \end{aligned}$$

The exit rule e_i ($1 \leq i \leq 5$) has the b_i-atom in r' as its head and the atoms in the set C_i as its body, where $C_5 = C_{5,1} \cup C_{5,2} \cup C_{5,3}$, cf. example 2.1.

A κ–sided normalized sirup can be **decomposed** by **counting techniques** into κ lower-dimensional sirups. Thus, the counting technique of section 1.2, which was restricted to special KNF sirups, namely linear DATALOG sirups in linear variable pattern, generalizes to arbitrary KNF sirups. The generalized decomposition approach yields a set of lower–dimensional sirups in a normal form called generalized form for transitive closure, which was introduced in [28], a set of non–recursive logic programs and one combining rule.

Definition 2.17 (GFTC) *A linear sirup LP is in* **generalized form for transitive closure (GFTC)**, *iff its recursive rule is of the form*

$$r : \ p(X, Y", Y, Z', m+1) : -p(X, Y", Y', Z', m), a(X, Y', Y, Z, V).$$

where subvectors with different names are disjoint.

Each recurrent predicate symbol of a KNF sirup gives rise to one GFTC sirup. The computed relations are joined via the numerical counting attribute. The simple structure of the GFTC guarantees a very efficient evaluation, since the well–known algorithms for evaluating ordinary transitive closures can be adapted to GFTC.

Example 2.18 *The KNF sirup $LP' = \{r', e_1, \ldots, e_5\}$ in example 2.16 for the 6–dimensional SLF sirup LP from example 2.1 can be decomposed into four 3–dimensional GFTC sirups (one for each recurrent predicate symbol b_i) like demonstrated for the sirup in linear variable pattern of section 1.2.*

2.4 Decomposition of Linear Murups

For sets of linear recursive rules defining the same intensional predicate symbol p an optimization technique can be developed based on the tracing of rule applications – like in the counting technique, cf. [26] – in an additional history argument position.

This technique is described here for the special case of a **multiple linear form (MLF)** murup, i.e. a linear DATALOG murup $LP = \{r_1, \ldots, r_n, e\}$ with several linear recursive rules r_i and one exit rule e with the following properties:

$$\begin{array}{llll} r_i: & p(X) & :- & p(U), a_{i,1}(U_{i,1}), \ldots, a_{i,n_i}(U_{i,n_i}). \\ e: & p(X) & :- & a_0(X). \end{array}$$

(MLF-1) the rules r_i are range–restricted,

(MLF-2) different body atoms $a_{i,j}(U_{i,j})$ do not share common arguments, their predicate symbols $a_{i,j}$ are unary or binary, i.e. $1 \leq |U_{i,j}| \leq 2$, $U_{i,j} \subseteq (X \cup U)$, $|U_{i,j} \cap X| \leq 1$, $|U_{i,j} \cap U| \leq 1$,

(MLF-3) $p(X)$ and $p(U)$ contain no constants and no repeated variables, shared variables occur in the same positions in both of them.

Extending the **expansion** formalism for SLF sirups, the iterate

$$r^{(i_1,\ldots,i_k)}: \quad p(X) \quad :- \quad p(V), reduced\_string(i_1, \ldots, i_k).$$

denotes the rule produced by successive SLD–resolution of the rules $r_{i_1}, \ldots, r_{i_k}$ starting with r_{i_1}. I.e., a further iterate $r^{(i_1,\ldots,i_k,i)}$ can be acchieved by resolving with a version of a rule r_i, where each variable symbol is superscribed by the tuple $(i_1, \ldots, i_k, i)$:

$$r_i\theta: \quad p(V) \quad :- \quad p(U), a_{i,1}(U_{i,1})\theta, \ldots, a_{i,n_i}(U_{i,n_i})\theta.$$

The **mixed AS–graph** $G = (V, E)$ of a murup LP is given by $V = \bigcup_{i=1}^n V_i$ and $E = \bigcup_{i=1}^n E_i$, where $G_i = (V_i, E_i)$ are the AS–graphs for the rules r_i. For each extensional predicate symbol there exists at most one incoming and one outgoing edge.

Thus, we can derive **counting rules** for LP like follows: for each pair α, β of argument positions of p connected by a directed path from α to β in G – regardless whether it is acyclic or cyclic, simple or not – and every γ, such that there are edges $(\beta \to a_{i,j})$, $(a_{i,j} \to \gamma) \in E$ we create a rule:

$$frontier_{\alpha,\gamma}(x_\alpha, u_\gamma, s \circ i) \quad :- \quad frontier_{\alpha,\beta}(x_\alpha, v_\beta, s), a_{i,j}(v_\beta, u_\gamma).$$

This rule means: if v_β can be derived from x_α with the rules $r_{i_1}, \ldots, r_{i_k}$ indicated by the sequence $s = (i_1, \ldots, i_k)$, and if u_γ can be derived from v_β with the predicate $a_{i,j}(v_\beta, u_\gamma)$, then u_γ can be derived from x_α with the rules indicated by the new sequence $s \circ i = (i_1, \ldots, i_k, i)$. Note, that the path from α to β does not occur in the generated rule. Only its existence is essential.

We need non–recursive exit rules only for the case $\alpha = \beta$ and $x_\alpha = v_\beta$:

$$frontier_{\alpha,\gamma}(x_\alpha, u_\gamma, i) \quad :- \quad a_{i,j}(x_\alpha, u_\gamma).$$

The counting program is completed by a rule which contains all the $frontier_{\alpha,\gamma}(x_\alpha, u_\gamma, s)$-predicates in its body and unifies them via their common history argument s:

$$p(X) \quad :- \quad p(U), \ldots, frontier_{\alpha,\gamma}(x_\alpha, u_\gamma, s), \ldots.$$

Example 2.19 *Given the linear murup $LP = \{r_1, r_2, e\}$, where*

$$r_1 : \quad p(x_1, x_2, x_3, x_4, x_5) \quad :- \quad p(u_1, u_2, u_3, u_4, x_5), a_{1,1}(x_1, u_1), a_{1,2}(x_3, u_3), a_{1,3}(x_2), a_{1,4}(x_4, u_4).$$
$$r_2 : \quad p(x_1, x_2, x_3, x_4, x_5) \quad :- \quad p(u_1, u_2, u_3, u_4, x_5), a_{2,1}(x_1, u_3), a_{2,2}(x_2, u_2), a_{2,3}(x_4), a_{2,4}(x_3, u_4).$$

and e is some exit rule.

The AS-graphs of r_1 and r_2 are mixed into one graph – shown in figure 5 – from which the counting rules are derived (only a subset of them is shown below):

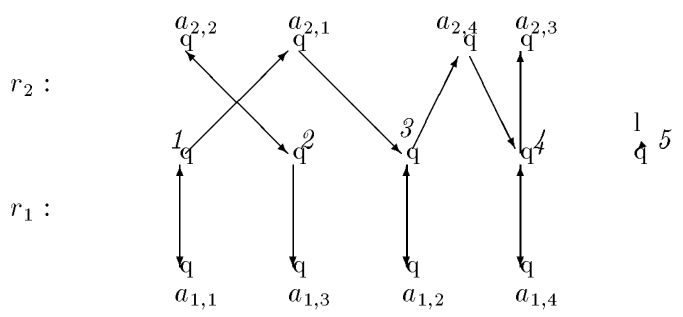

Figure 5: Mixed AS–graph of LP

$$\begin{aligned}
frontier_{1,1}(x_1, u_1, 1) \quad &:- \quad a_{1,1}(x_1, u_1).\\
frontier_{1,1}(x_1, u_1, s \circ 2) \quad &:- \quad frontier_{1,1}(x_1, v_1, s), a_{1,1}(v_1, u_1).\\
\\
frontier_{1,3}(x_1, u_3, 2) \quad &:- \quad a_{2,1}(x_1, u_3).\\
frontier_{1,3}(x_1, u_3, s \circ 2) \quad &:- \quad frontier_{1,1}(x_1, v_1, s), a_{2,1}(v_1, u_3).\\
frontier_{1,3}(x_1, u_3, s \circ 1) \quad &:- \quad frontier_{1,3}(x_1, v_3, s), a_{1,2}(v_3, u_3).\\
\\
frontier_{1,4}(x_1, u_4, s \circ 2) \quad &:- \quad frontier_{1,3}(x_1, v_3, s), a_{2,4}(v_3, u_4).\\
\\
p(x_1, x_2, x_3, x_4, x_5) \quad &:- \quad p(u_1, u_2, u_3, u_4, x_5), \ldots, frontier_{i,j}(x_i, u_j, s), \ldots
\end{aligned}$$

The technique can be extended to work for linear murups without the property **(MLF-2)**. In this more general case the relations $frontier_{\alpha,\gamma}$ link sets α and γ of argument positions.

2.5 Two–Dimensional SLF Sirups and CFG Programs

A lot of 2-dimensional SLF sirups can be normalized into a CFG program. The CFG program representation can then be used in an evaluation based on the transition graph, cf. section 3.

Example 2.20 *The rules r_1, r_2 are* **one–sided** *and their normal form is a CFG program, the rules r_3, r_4 are* **one–sided**, *but their normal form is not a CFG program:*

$$\begin{array}{llll}
r_1: & p(x_1, x_2) & :- & p(u_1, u_2), a_1(x_1, u_1), a_2(x_2), a_3(u_2). \\
r_2: & p(x_1, x_2) & :- & p(u_1, x_2), a_1(x_1, u_1). \\
r_3: & p(x_1, x_2) & :- & p(u_1, u_2), a_1(x_1, x_2, u_1), a_2(u_2). \\
r_4: & p(x_1, x_2) & :- & p(x_2, u_2), a_1(x_2, u_2), a_2(x_1).
\end{array}$$

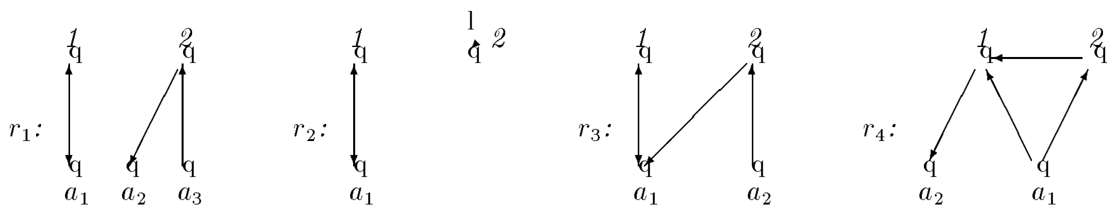

Figure 6: AS–graphs of one–sided CFG programs

For all of the **two–sided** *rules the normal form is a CFG program:*

$$\begin{array}{llll}
r_5: & p(x_1, x_2) & :- & p(x_2, u_2), a_1(x_1, u_2). \\
r_6: & p(x_1, x_2) & :- & p(u_1, u_2), a_1(x_1, u_2), a_2(x_2, u_1). \\
r_7: & p(x_1, x_2) & :- & p(u_1, u_2), a_1(x_1, u_1), a_2(x_2, u_2).
\end{array}$$

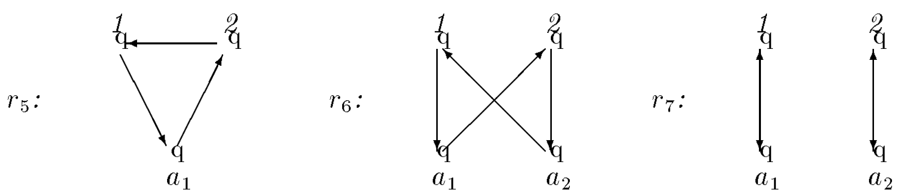

Figure 7: AS–graphs of two–sided CFG programs

The corresponding AS–graphs are shown in figures 6 and 7.

3 Graph–Based Evaluation of CFG Programs

Expanding a CFG program naturally corresponds to deriving strings from an associated context–free grammar. We introduce a technique that transforms a finite subset of compiled formulae in the expansion of a CFG program into a transition graph based on the prefixes of the corresponding strings. This graph guides the ordered, complete, but non–redundant generation of answer tuples for the CFG program. Common subexpressions between different compiled formulae yield the same path in the graph.

The resulting interformula optimization is a generalization of the well–known wavefront techniques, that use intermediate results from one iteration in following iterations.

3.1 Transition Graphs for CFG Programs

For any CFG program $\mathcal{P}$ and predicate symbol p in $\mathcal{P}$ we associate a (canonical) context–free grammar $\mathcal{G}_{\mathcal{P}}$ and a transition graph $G_{\mathcal{L}}$ for a finite subset $\mathcal{L}$ of the generated language. This language corresponds to some compiled formulae in the expansion of $\mathcal{P}$. The transition graph contains a node for each prefix of a word in the language and a directed edge (s, s') between two prefixes of words where $s' = s \circ q$ for a predicate symbol q. This edge is labelled by q and denoted by $(s \xrightarrow{q} s')$.

Definition 3.1 (Transition Graph for CFG Program) *Given a CFG program $\mathcal{P}$ and a predicate symbol p in $\mathcal{P}$.*

(i) *The* **alphabet** $\Sigma_{\mathcal{P}}$ *for $\mathcal{P}$ is given by the set of predicate symbols used in $\mathcal{P}$:*

- $\Sigma_{\mathcal{P}} = \{p_i | (\ p_0(X)\ :-\ p_1(X_1), \ldots, p_n(X_n)\ ) \in \mathcal{P}, 0 \leq i \leq n\ \}$.

(ii) *The* **context-free grammar** $\mathcal{G}_{\mathcal{P}} = (V, T, P, S)$ *for $\mathcal{P}$ and p is given by:*

- *variables and terminals in* $V = T = \Sigma_{\mathcal{P}}$,
- *productions in* $P = \{p_0 \to p_1 \ldots p_n | (\ p_0(X)\ :-\ p_1(X_1), \ldots, p_n(X_n)\ ) \in \mathcal{P}\ \}$,
- *the start symbol* $S = p$.

The language generated by $\mathcal{G}_{\mathcal{P}}$ is denoted by $L(\mathcal{G}_{\mathcal{P}})$. The language generated by $\mathcal{G}_{\mathcal{P}}$ with derivation trees of depth at most $n \in I\!N_0$ is denoted by $L_n(\mathcal{G}_{\mathcal{P}})$.

(iii) *The* **transition graph** $G_{\mathcal{L}} = (V_{\mathcal{L}}, E_{\mathcal{L}})$ *for some finite sublanguage $\mathcal{L} \subseteq L(\mathcal{G}_{\mathcal{P}})$ is given by:*

- $V_{\mathcal{L}} = prefix(\mathcal{L})$,
- $E_{\mathcal{L}} = \{(v_1, v_2) | v_1, v_2 \in V, (\exists c \in \Sigma_{\mathcal{P}} : v_2 = v_1 \circ c)\}$,

where $s_1 \circ s_2$ denotes the concatenation of two strings s_1 and s_2.

Example 3.2 *Given the CFG program $\mathcal{P}$:*

$$\begin{array}{llll} r_1 : & p(x_1, x_6) & :- & a(x_1, x_2), p(x_2, x_3), b(x_3, x_4), p(x_4, x_5), c(x_5, x_6). \\ r_2 : & p(x_1, x_4) & :- & a(x_1, x_2), p(x_2, x_3), c(x_3, x_4). \\ r_3 : & p(x_1, x_4) & :- & e(x_1, x_2), p(x_2, x_3), c(x_3, x_4). \end{array}$$

The transition graph $G_{\mathcal{L}}$ for the language $\mathcal{L} = \{apbpc, apc, epc\}$ is given by figure 8.

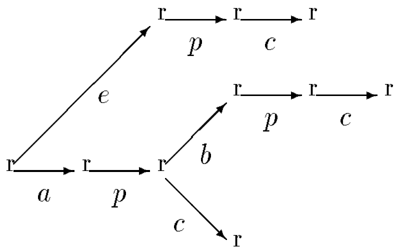

Figure 8: Transition graph

The well–known **minimization** algorithm of **Huffman and Moore**, cf. [16], yields a minimal transition graph $G^*_{\mathcal{L}} = (V^*_{\mathcal{L}}, E^*_{\mathcal{L}})$, whose edges are labelled by predicate symbols in $\mathcal{P}$. There is exactly one node v_ϵ without predecessor, called initial node, corresponding to the empty string ϵ, and exactly one node v_a without successor, called answer node.

Example 3.3 (continued) *Minimization by the algorithm of Huffman and Moore yields the graph of figure 9.*

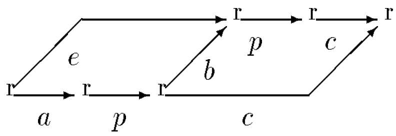

Figure 9: Minimal transition graph for $L_1(\mathcal{G}_{\mathcal{P}})$

The minimal transition graph $G^*_{\mathcal{L}}$ for the language

$$L_2(\mathcal{G}_{\mathcal{P}}) = \{apbpc, apc, epc, aapbpcbpc, \ldots, apbepcc, aapbpcc, aapcc, aepcc, eapbpcc, eapcc, eepcc\},$$

which corresponds to the derivation trees of depth at most 2 *of the above program, is given the graph of figure 10.*

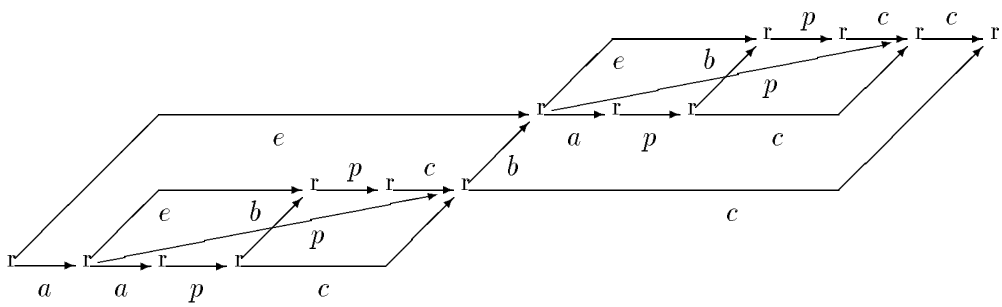

Figure 10: Minimal transition graph for $L_2(\mathcal{G}_{\mathcal{P}})$

The language $L_n(\mathcal{G}_\mathcal{P})$ contains exponentially many strings. This shown in appendix B even for the subset $\mathcal{P}' = \{r_1\}$ of $\mathcal{P}$ containing just the rule r_1. Thus, the minimization of the induced transition graph is very important as soon as a larger n is used.

Given a CFG program $\mathcal{P}$, a predicate symbol p in $\mathcal{P}$ and any finite language $\mathcal{L} \subseteq L(\mathcal{G}_\mathcal{P})$. According to the minimal transition graph $G^*_\mathcal{L} = (V^*_\mathcal{L}, E^*_\mathcal{L})$ we define a CFG program $\mathcal{P}_\mathcal{L}$ consisting of the following rules. For each edge $(u \xrightarrow{q} v) \in E^*_\mathcal{L}$ we define a CFG rule $r_{u,v}$, if $u \neq v_\epsilon$, and a CFG rule r_v, if $u = v_\epsilon$. For the answer node v_a we introduce the answer rule r_a:

$$\begin{array}{rlcl} r_{u,v}: & p_v(x_1, x_3) & :- & p_u(x_1, x_2), q(x_2, x_3). \\ r_v: & p_v(x_1, x_3) & :- & q(x_1, x_3). \\ r_a: & p(x_1, x_2) & :- & p_a(x_1, x_2). \end{array}$$

Then for each language $\mathcal{L} = L_n(\mathcal{G}_\mathcal{P})$ the CFG program $\mathcal{P}_\mathcal{L}$ is p–equivalent to $\mathcal{P}$. The relation corresponding to a predicate symbol p_v in $\mathcal{P}_\mathcal{L}$ will be denoted by $R(v)$ for each node v in the minimal transition graph.

The graph–based evaluation embodies the following **query processing techniques** to reduce several types of redundancy compared to the separate processing of the strings, cf. [12]:

- **(i)** **Frontier relations** avoid the repeated processing of portions of strings from different derivation tree depths. They are updated in each iteration of the evaluation algorithm.
- **(ii)** **Δ–iteration** sends only the newly derived tuples of a node relation to the nodes successors.
- **(iii)** **Common subexpessions** are evaluated only once for the strings of the same derivation tree depth.
- **(iv)** **Tuning** assures that a set of q–tuples fetched from secondary storage is exploited at all q–edges. This improves the file accessing behavior, since otherwise a relation residing on secondary storage would – apart from cashing – be repeatedly fetched for several successive iterations.

The evaluation algorithm GENERALIZED_TC determines the relations $R(v)$ for each node v in the minimal transition graph $G^*_\mathcal{L} = (V^*_\mathcal{L}, E^*_\mathcal{L})$ of a CFG program. following the rules in $\mathcal{P}_\mathcal{L}$.

Algorithm GENERALIZED_TC ($V^*_{\mathcal{L}}, E^*_{\mathcal{L}}$)
{ initialization }
$i := 0;\ P := \emptyset;$
for all $v \in V^*_{\mathcal{L}}$ **do**
    $R_0(v) := \emptyset;$
for all $(v_\epsilon \xrightarrow{q} v) \in E^*_{\mathcal{L}}$ **do**
    $R_0(v) := Q;$
{ iteration }
while (there is $v \in V^*_{\mathcal{L}}$: $\Delta R_i(v) \neq \emptyset$) **do**
    $i := i + 1;$
    for all $(u \xrightarrow{q} v) \in E^*_{\mathcal{L}},\ q \neq p$ **do**
        $\Delta R_i(v) := (\Delta R_{i-1}(u) \circ Q) \setminus R_{i-1}(v);$
        $R_i(v) := R_{i-1}(v) \cup \Delta R_i(v));$
      end
    $\Delta P := \Delta R_{i-1}(v_a);$
    $P := P \cup \Delta P;$
    for all $(u \xrightarrow{p} v) \in E^*_{\mathcal{L}}$ **do**
        $\Delta R_i(v) := (\ (\Delta R_{i-1}(u) \circ P) \cup (R_{i-1}(u) \circ \Delta P)\ ) \setminus R_{i-1}(v);$
        $R_i(v) := R_{i-1}(v) \cup \Delta R_i(v));$
      end
    end

The algorithm uses the composition operator defined by $A \circ B = \Pi^{(4)}_{1,4}(\ A \bowtie_{A2=A3} B\ )$ which is supported efficiently from relational algebra.

3.2 Transition Graphs for General Logic Programs

Firstly, we observe that every logic rule can be mapped to a linear relational algebra tree, cf. figure 11,

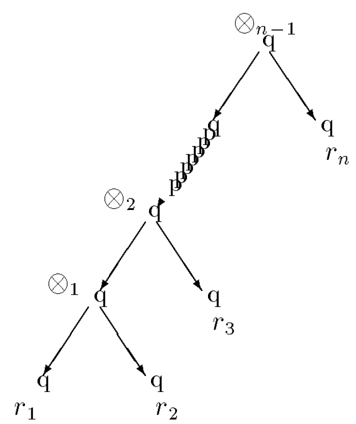

Figure 11: Linear relational algebra tree

which corresponds to the following relational algebra expression:

$$(\ldots((\ r_1 \otimes_1 r_2\ ) \otimes_2 r_3\ ) \ldots \otimes_{n-1} r_n\ ),$$

where r_i denotes a relation and $\otimes_i$ denotes an algebraic operation. Thus, we consider the transition graph for the finite language $\mathcal{L}$ containing the strings:

$$r_1 \otimes_1 r_2 \otimes_2 r_3\ \ldots\ \otimes_{n-1} r_n$$

Obviously, the evaluation algorithm GENERALIZED_TC can be extended to work for these extended transition graphs, too.

Example 3.4 *The logic rule*

$$p(x_1, x_2, x_3) : -p(u_1, u_2, u_3), a(x_1, x_2, u_2), b(x_3, u_1), c(u_2, u_3).$$

can be transformed to the following relational algebra expression:

$$\Pi^{(5)}_{1,2,4}(\ \Pi^{(7)}_{1,2,5}(\ \Pi^{(5)}_{1,2,3,5}(\ a \bowtie_{A3=A4} c\ )\ \bowtie_{A3=A6,A4=A7} p\ )\ \bowtie_{A3=A5} b\ )$$

Example 3.5 *The logic program*

$$\begin{array}{lcl} p(x_3, x_6) & :- & a(x_1, x_2), p(x_2, x_3), b(x_1, x_4), p(x_5, x_4), c(x_5, x_6). \\ p(x_1, x_4) & :- & a(x_1, x_2), p(x_2, x_3), c(x_4, x_3). \\ p(x_2, x_4) & :- & e(x_1, x_2), p(x_1, x_3), c(x_3, x_4). \end{array}$$

with binary symbols can be transformed to the following language $\mathcal{L}$ of relational algebra expressions:

$$\mathcal{L} = \{\ a \circ p \bullet b \star p \circ c,\ \ a \circ p \star c,\ \ e \bullet p \circ c\ \}$$

where $a \circ b\ =\ \Pi^{(4)}_{1,4}(\ a \bowtie_{A2=A3} b\ )$, $a \bullet b\ =\ \Pi^{(4)}_{2,4}(\ a \bowtie_{A1=A3} b\ )$, $a \star b\ =\ \Pi^{(4)}_{1,3}(\ a \bowtie_{A2=A4} b\ )$.

The minimized transition graph for $\mathcal{L}$ is given by figure 12.

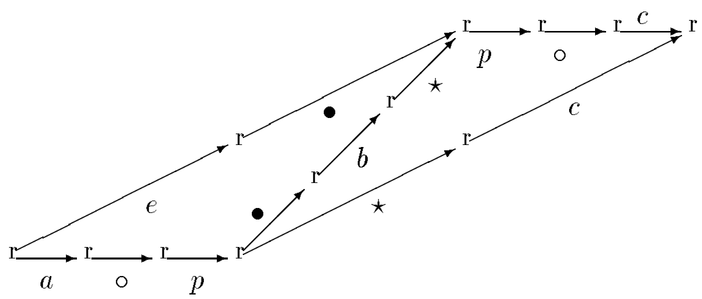

Figure 12: Minimal transition graph for general logic program

4 Conclusion

The decomposition approach for CKNF sirups can be formulated equivalently as a **wavefront technique**, cf. [28], based on so-called composition operators, cf. [29]. These associative operators induce powers of the extensional a_i-databases that can be computed by a logarithmic number of database operations.

The expansion of an SLF sirup w.r.t. an arbitrary goal $G_0 = p(X')$ including constant symbols or repeated variables (instead of the goal $p(X)$ formed by the head of its recursive rule) corresponds to a query which poses a **selection** to the derived p-database. It should be possible to get optimized query evaluation plans for these queries too.

Transition graphs are a new concept for the iterative processing of recursive logic programs. Again, the resulting interformula optimization for the processing of the strings is a wavefront technique that uses intermediate results from one iteration in the following iterations.

We are currently testing which sublanguage $\mathcal{L}$ of the context–free language $\mathcal{L}(\mathcal{G}_{\mathcal{P}})$ generated by a CFG program $\mathcal{P}$ is optimal for generating a corresponding transition graph $\mathcal{G}_{\mathcal{L}}$. Furthermore, the language $\mathcal{L}$ can be considered over different basic alphabets Σ – e.g. a string $p_i \circ p_j$ can be considered as a single symbol – which may yield more efficiently evaluable transition graphs for $\mathcal{P}$. Finally, it would be desirable to incorporate binding propagation, cf. the **magic sets** method, for a given query goal $p(c, Y)$, where c is a constant, into the graph–based evaluation.

Appendix

A AS–Graphs and the Sidedness of SLF Sirups

We will use the following **notations for paths** in AS–graphs. A path π from a node v_0 to a node v_m is given by a sequence $\pi = (e_1, \ldots, e_m)$ $(m \in I\!N_+)$ of edges $e_i = (v_{i-1} \xrightarrow{l_i} v_i)$ $(1 \leq i \leq m)$. The length of π is $l(\pi) = \sum_{i=1}^{m-1} l(e_i)$, the start and end node of π is $\alpha(\pi) = v_0$ and $\omega(\pi) = v_m$, respectively. π is a cycle, iff $v_0 = v_m$. The type of π is given by $(v_0 \xrightarrow{l(\pi)} v_m)$. The $j-th$ subpath $(0 \leq j \leq m)$ of π is given by $\pi_{1..j} = (e_1, ..., e_j)$, where $\pi_{1..0} = ()$ is an empty sequence of the length $l(\pi_{1..0}) = 0$.

By considering special paths in the AS–graph, the following **questions** will be answered:

- Which **intensional** atoms $p(U^j)$ $(j \in I\!N_+)$ share arguments ?
- Which **extensional** atoms $a_i(U_i)\theta_j$ $(1 \leq i \leq n,\ j \in I\!N_+)$ share arguments or are connected within $reduced\_string(m)$ $(m \in I\!N_+ \cup \{\infty\})$?

The first question is answered by the following theorem concerning the S–graph.

Lemma A.1 *Given an SLF sirup. The positions i_1 in U^{j_1} and i_2 in U^{j_2} are equal, iff there is a path π of the type $(i_1 \xrightarrow{j_2-j_1} i_2)$ in the S–graph, such that $j_1 + l_{min}(\pi) \geq 0$, where $l_{min}(\pi) = min\{l(\pi_{1..j}) | 0 \leq j \leq m\}$.*

The answer to the second question refers to paths in the AS–graph that have certain properties concerning the lengths of all subpaths and of those subpaths that end at an extensional predicate symbol.

Lemma A.2 *Given an SLF sirup.*

(i) *Two different atoms $a_{i_1}(U_{i_1})\theta_{j_1}$, $a_{i_2}(U_{i_2})\theta_{j_2}$ share a common argument, iff there is a path $\pi = (e_1, \ldots, e_l)$ of the type $(a_{i_1} \xrightarrow{j_2 - j_1} a_{i_2})$ in the AS–graph, such that*
- *$l \geq 2$ and $\pi' = (e_2, \ldots, e_{l-1})$ either is a path in the S–graph or $\pi' = ()$, and*
- *$j_1 + l_{min}(\pi) \geq 1/2$.*

(ii) *Two different atoms $a_{i_1}(U_{i_1})\theta_{j_1}$, $a_{i_2}(U_{i_2})\theta_{j_2}$ are connected within reduced_string(m), for $m \in I\!N_+ \cup \{\infty\}$, iff there is a path π of type $(a_{i_1} \xrightarrow{j_2 - j_1} a_{i_2})$ in the AS–graph, such that*
- *$j_1 + l_{min}(\pi) \geq 1/2$ and*
- *$j_1 + a_{max}(\pi) \leq m$ (for $m = \infty$ this poses no restriction on π),*
where $a_{max}(\pi) = max\{l(\pi_{1..j}) | \omega(\pi_{1..j}) \in \{a_1, \ldots, a_n\}, 0 \leq j \leq m\}$.

The exact proofs of lemma A.1 and A.2 require a detailed analysis of the passing of variable bindings during the SLD–resolution process. The interested reader is referred to [Sei 89]. The proof of lemma A.2 is based on lemma A.1 and the fact, that two different extensional atoms $a_{i_1}(U_{i_1})\theta_{j_1}$ and $a_{i_2}(U_{i_2})\theta_{j_2}$ share a common argument x, iff they share x with two intensional atoms $p(U^{k_1})$ and $p(U^{k_2})$ for some $k_1 \in \{j_1, j_1 - 1\}$ and $k_2 \in \{j_2, j_2 - 1\}$, respectively.

Theorem A.3 (Sidedness of SLF sirups) *An SLF sirup LP is κ-sided for $\kappa = \sum_{\nu=1}^{k} l_\nu$. Each of its iterates LP^m ($m \in I\!N_+$) is κ-sided, too.*

Sketch of the proof:

Recurrent predicate symbols $a_i, a_j \in R_\nu$ ($1 \leq \nu \leq k$) in the same recurrent class R_ν are connected in the AS–graph and there exists a minimal, non-negative distance $m_{i,j} \in < 0, l_\nu - 1 >$ from a_i to a_j in the AS–graph. Isolated predicate symbols $a_i, a_j \in R_0$ can be connected in the AS–graph. If they are so, then there exists a unique distance $m_{i,j}$ from a_i to a_j.

The connectivity information about the extensional predicate symbols a_i in the AS–graph can be translated to connectivity information about the extensional atoms $a_i(U_i)\theta_j$ in the expansion. The following **equivalence relation** is a useful coarsening of the connected components, i.e. connected atoms can be proven to be equivalent:

$$(a_{i_1}(U_{i_1})\theta_{j_1} \equiv a_{i_2}(U_{i_2})\theta_{j_2}), \text{ iff}$$
$$(a_{i_1}, a_{i_2} \in R_\nu, 1 \leq \nu \leq k \text{ and } j_2 = j_1 + m_{i_1,i_2} \pmod{l_\nu}) \text{ or } (a_{i_1}, a_{i_2} \in R_0).$$

There are $\sum_{\nu=1}^{k} l_\nu$ equivalence classes containing recurrent atoms. It remains to be shown that each of them contains exactly one **infinite connected component**, and that the equivalence class consisting of all the isolated atoms contains only bounded connected components.

We fix an arbitrary set $\wp$ of paths in the AS–graph that contains one connecting path $\pi_{i,j}$ of type $(a_i \xrightarrow{m_{i,j}} a_j)$ for each pair of recurrent predicate symbols $a_i, a_j \in R_\nu$ $(1 \leq \nu \leq k)$ in the same recurrent class R_ν and for each pair $a_i, a_j \in R_0$ of isolated predicate symbols which are connected in the AS–graph. We define $\tau_1 = max\{a_{max}(\pi) | \pi \in \wp\}$, $\tau_2 = max\{-l_{min}(\pi) | \pi \in \wp\} - 1/2$. For all $m \geq \tau_1 + \tau_2$ – based on lemma A.2 – the following can be shown:

For $I = < \tau_2 + 1, m - \tau_1 >$ and $I' = < 1, \tau_2 > \cup < m - \tau_1 + 1, m >$ we partition the set $reduced\_string(m)$ into two parts and investigate connectedness within $reduced\_string(m)$. Given $a_{i_1}(U_{i_1})\theta_{j_1}$ and $a_{i_2}(U_{i_1})\theta_{j_2}$ in

$$reduced\_string(I) = \{a_i(U_i)\theta_j \mid j \in I \}.$$

- For $a_{i_1} \in R$, it holds: $a_{i_1}(U_{i_1})\theta_{j_1}$ and $a_{i_2}(U_{i_1})\theta_{j_2}$ are connected in $reduced\_string(m)$, iff $a_{i_2}(U_{i_1})\theta_{j_1} \equiv a_{i_2}(U_{i_2})\theta_{j_2}$.
- For $a_{i_1} \in R_0$, it holds: if $a_{i_1}(U_{i_1})\theta_{j_1}$ is connected to $a_{i_2}(U_{i_2})\theta_{j_2}$ in $reduced\_string(m)$, then $a_{i_2} \in R_0$.

The complement set

$$reduced\_string(I') = \{a_i(U_i)\theta_j \mid j \in I' \}$$

contains $\tau_1 + \tau_2$ elements, independently of m.

Thus, equivalent recurrent atoms in $reduced\_string(I)$ are contained in the same infinitely growing connected component C in $reduced\_string(m)$. The recurrent atoms in $reduced\_string(I')$ can be contained in C, too. They cannot form another infinitely growing connected component C', since $reduced\_string(I')$ contains only $\tau_1 + \tau_2$ elements, for all m. The isolated atoms in $reduced\_string(I)$ are contained in bounded connected components C in $reduced\_string(m)$, each containing at most one a_i–atom for each $a_i \in R_0$, i.e. of size at most $|R_0|$.

Hence, LP is κ-sided, since $\kappa = \sum_{\nu=1}^{k} l_\nu$. It is easy to show that LP^m ($m \in I\!N_+$) is κ-sided, too. □

B Expansions of CFG Programs

Lemma B.1 *Let b_n be the number of strings in the language $L_n(\mathcal{G}_{\mathcal{P}'})$ generated by $\mathcal{G}_{\mathcal{P}'}$ with derivation trees of depth at most $n \in I\!N_0$ for the CFG program $\mathcal{P}' = \{r_1\}$ with the bilinear recursive rule*

$$r_1 : \quad p(x_1, x_6) \ :- \ a(x_1, x_2), p(x_2, x_3), b(x_3, x_4), p(x_4, x_5), c(x_5, x_6).$$

Then it holds

$$\begin{array}{ll} (1) & b_n = b_{n-1}^2 + 1, \\ (2) & b_n = \prod_{i=1}^{n-1}(b_i + b_{i-1}), \\ (3) & 2^{2^{n-1}} < b_n < 2.4^{2^{n-1}}, \forall n \geq 3. \end{array}$$

Proof:

Let $a_n = b_n - b_{n-1}$ ($n \in I\!N_0$) be the number of strings in $L_n(\mathcal{G}_{\mathcal{P}'})$ generated with derivation trees of depth n, where $b_{-1} = 0$ for technical reasons.

The strings of depth n are derived from a string of depth $n-1$ and an arbitrary string of depth at most $n-2$ or from two strings of depth $n-1$. Obviously $b_0 = 1$ and $b_1 = 2$.

Thus

$$\begin{aligned} a_n &= a_{n-1} \cdot b_{n-2} + b_{n-2} \cdot a_{n-1} + a_{n-1} \cdot a_{n-1} \\ &= 2 \cdot a_{n-1} \cdot b_{n-2} + a_{n-1}^2 \\ &= 2 \cdot (b_{n-1} - b_{n-2}) \cdot b_{n-2} + (b_{n-1} - b_{n-2})^2 \\ &= (2 \cdot b_{n-2} + b_{n-1} - b_{n-2}) \cdot (b_{n-1} - b_{n-2}) \\ &= b_{n-1}^2 - b_{n-2}^2. \end{aligned}$$

By induction on n we get

$$\begin{aligned} b_n - b_{n-1} &= a_n \\ &= (b_{n-1} + b_{n-2}) \cdot (b_{n-1} - b_{n-2}) \\ &= (b_{n-1} + b_{n-2}) \cdot (b_{n-2} + b_{n-3}) \cdot \ldots \cdot (b_1 + b_0) \cdot \overbrace{(b_1 - b_0)}^{=1} \\ &= (b_{n-1} + b_{n-2}) \cdot \ldots \cdot (b_1 + b_0) \\ &= \prod_{i=1}^{n-1} (b_i + b_{i-1}). \end{aligned}$$

Since

$$b_n - b_{n-1} = a_n = b_{n-1}^2 - b_{n-2}^2,$$

again by induction on n, we have

$$b_n - b_{n-1}^2 = b_{n-1} - b_{n-2}^2 = \ldots = b_1 - b_0^2 = 1,$$

and

$$b_n = b_{n-1}^2 + 1.$$

Thus it holds for all $n \geq 3$:

$$b_{n-1}^2 < b_n < 1.21 \cdot b_{n-1}^2 = (1.1 \cdot b_{n-1})^2,$$

which, again by an induction on n, implies that

$$b_2^{2^{n-2}} < b_n < (1.1 \cdot b_2)^{2^{n-2}}.$$

Since $b_2 = 5$, we have $2^{2^{n-1}} < b_2^{2^{n-2}}$, $(1.1 \cdot b_2)^{2^{n-2}} < 2.4^{2^{n-1}}$, and

$$2^{2^{n-1}} < b_n < 2.4^{2^{n-1}}.$$

Thus, the above equations hold. □

For example it holds $b_0 = 1, b_1 = 2, b_2 = 5, b_3 = 26, b_4 = 677$.

Acknowledgments

The author would like to thank his colleagues Hans Argenton and Helmut Thöne for the many useful comments that they have provided.

References

[1] **R. Agrawal, P. Devanbu:** Moving Selections into Linear Least Fixpoint Queries, Proc. Intl. Conf. on Data Engineering, 1988, pp. 452-461.

[2] **F. Bancilhon, D. Maier, Y. Sagiv, J. Ullman:** Magic Sets and Other Strange Ways to Implement Logic Programs, Proc. PODS 1986, pp. 1-15.

[3] **F. Bancilhon, R. Ramakrishnan:** An Amateur's Introduction to Recursive Query Processing Strategies, Proc. ACM SIGMOD 1986, pp. 16-52.

[4] **C. Beeri, R. Ramakrishnan:** On the Power of Magic, Poc. PODS 1987, pp. 269-283.

[5] **J. Biskup, H. Stiefeling:** Transitive Closure Algorithms for Very Large Databases, Proc. Intl. Workshop on Graph-Theoretic Concepts in Computer Science 1988 (WG'88), LNCS 344, Springer, 1989, pp. 122-147.

[6] **S. Ceri, G. Gottlob, L. Tanca:** Logic Programming and Databases, Springer, 1990.

[7] **S.K. Das:** Deductive Databases and Logic Programming, Addison–Wesley, 1992.

[8] **G. Dong:** Datalog Expressiveness of Chain Queries: Grammar Tools and Characterization, Proc. PODS 1992, pp. 81-90.

[9] **H. Gallaire, J. Minker, J. Nicolas:** Logic and Databases: A Deductive Approach, ACM Computing Surveys, vol. 16(2), 1984, pp. 153-185.

[10] **G. Grahne, S. Sippu, E. Soisalon-Soininen:** Efficient Evaluation for a Subset of Recursive Queries, Proc. PODS 1987, pp. 284-293.

[11] **U. Güntzer, W. Kiessling, R. Bayer:** On the Evaluation of Recursion in (Deductive) Database Systems by Efficient Differential Fixpoint Iteration, Proc. Intl. Conf. on Data Engineering 1987, pp. 120-129.

[12] **J. Han, L. Henschen:** Handling Redundancy in the Processing of Recursive Database Queries, Proc. ACM SIGMOD 1987, pp. 73-81.

[13] **J. Han, H. Lu:** Some Performance Results on Recursive Query Processing in Relational Database Systems, Proc. Intl. Conf. on Data Engineering 1986, pp. 533-554.

[14] **R. Haddad, J. Naughton:** Counting Methods for Cyclic Relations, Proc. PODS 1988, pp. 333-340.

[15] **L. Henschen, S. Naqvi:** On Compiling Queries in Recursive First-Order Data Bases, JACM, vol. 31(1), 1984, 47-85.

[16] **J.E. Hopcroft, J.D. Ullman:** Introduction to Automata Theory, Languages and Computation, Addison–Wesley, 1979.

[17] **M. Houtsma, P. Apers:** Algebraic Optimization of Recursive Queries, Data & Knowledge Engineering, vol. 7, 1992, pp. 299-325.

[18] **Y. Ioannidis, E. Wong:** An Algebraic Approach to Recursive Inference, in [20], pp. 295-309.

[19] **H. Jagadish, R. Agrawal, L. Ness:** A Study of Transitive Closure as a Recursion Mechanism, Proc. ACM SIGMOD 1987, pp. 331-344.

[20] **L. Kerschberg:** Expert Database Systems, Proc. First Intl. Workshop 1986, Benjamin-Cummings, Menlo Park, CA, 1987.

[21] **J.W. Lloyd:** Foundations of Logic Programming, Springer, second edition, 1987.

[22] **J. Naughton:** One-Sided Recursions, Proc. PODS 1987, pp. 340-348.

[23] **J. Naughton:** Minimizing Function-Free Recursive Inference Rules, JACM, vol. 36(1), 1989, pp. 69-91.

[24] **J. Naughton, Y. Sagiv:** A Decidable Class of Bounded Recursions, Proc. PODS 1987, pp. 227-236.

[25] **Y. Sagiv:** Optimizing Datalog Programs, Proc. PODS 1987, pp. 349-362.

[26] **D. Sacca, C. Zaniolo:** The Generalized Counting Method for Recursive Logic Queries, Proc. ICDT'86.

[27] **D. Sacca, C. Zaniolo:** Magic Counting Methods, Proc. ACM SIGMOD 1987, pp. 49-49.

[28] **D. Seipel:** Decomposition in Database and Knowledge-Base Systems, Doctoral Dissertation, University of Würzburg, 1989.

[29] **S. Sippu, E. Soisalon-Soininen:** A Generalized Transitive Closure for Relational Queries, Proc. PODS 1988, pp. 325-332. (and: Research Report 183, Univ. Karlsruhe, Dept. of CS, 1987).

[30] **J.D. Ullman**: Principles of Database and Knowledge-Base Systems, Volume I/II, Computer Science Press, 1988/89.

[31] **C. Youn, L. Henschen, J. Han:** Classification of Recursive Formulas in Deductive Databases, Proc. ACM SIGMOD 1988, pp. 320-328.