Expressing and Supporting Efficiently Greedy Algorithms as Locally Stratified Logic Programs

CARLO ZANIOLO
University of California, Los Angeles
E-mail: zaniolo@cs.ucla.edu

submitted 29 April 2015; accepted 5 June 2015

Abstract

The problem of expressing and supporting classical greedy algorithms in Datalog has been the focus of many significant research efforts that have produced very interesting solutions for particular algorithms. But we still lack a general treatment that characterizes the relationship of greedy algorithms to non-monotonic theories and leads to asymptotically optimal implementations. In this paper, we propose a general solution to this problem. Our approach begins by identifying a class of locally stratified programs that subsumes XY-stratified programs and is formally characterized using the Datalog$_{1S}$ representation of numbers. Then, we propose a simple specialization of the iterated fixpoint procedure that computes efficiently the perfect model for these programs, achieving optimal asymptotic complexities for well-known greedy algorithms.This makes possible their efficient support in Datalog systems.

KEYWORDS: Horn Clauses, Datalog, Aggregates, Greedy Algorithms.

1 Introduction

Due to the emergence of many important application areas, we are now experiencing a major resurgence of interest in Datalog for parallel and distributed programming (Hellerstein 2010; Abiteboul et al. 2011) (Gottlob et al. 2011; Abiteboul et al. 2011). This include exploring parallel execution of recursive queries in the MapReduce framework (Afrati et al. 2011) and on multicore machines (Yang et al. 2015), and in Data Stream Management Systems (Zaniolo 2011). The abundance of new applications underscores the need to tackle and solve crucial Datalog problems that have remained open for a long tim—starting with algorithms that require aggregates in recursive rules that provided the subject of much previous work (Zaniolo et al. 1997; Greco and Zaniolo 2001a; Mumick et al. 1990; Kolaitis 1991; Mumick and Shmueli 1995; Ross and Sagiv 1997). In this context, a major step forward was accomplished recently with the introduction of monotonic aggregates (Mazuran et al. 2013; Shkapsky et al. 2013). Monotonic aggregates, however, cannot address the problem of formulating and supporting efficiently greedy algorithms, a difficult challenge that provided the focus of much previous research, including (Greco et al. 1992; Greco and Zaniolo 1998; Greco and Zaniolo 2001b). Their work provided a stable-model characterization for specific greedy algorithms but did not develop a general theory and efficient solutions for such programs. In this paper, we achieve a general solution by showing that greedy algorithms can be expressed quite naturally as locally stratified programs that are conducive to a very efficient implementation—i.e., one having the same asymptotic complexity as that achievable using procedural languages and specialized data structures. This is a very encouraging result, given that optimal performance is not easily achievable for algorithms expressed in the concise and elegant formalism of declarative logic, i.e., without having to specify detailed operational steps and special data structures in many pages of procedural code. Furthermore, many intractability results obtained for locally stratified logic programs (Palopoli 1992) underscore the difficulty of using them to express and support low-complexity algorithms. But in this paper we show that there is a natural correspondence between greedy algorithm and a special subclass of locally stratified programs, which we will call strictly stratified temporal programs, that overcome these difficulties.

In the next section we recall the basic notions of local stratification, and iterated fixpoint, and then, in Section 3, we introduce a class of of programs that are locally stratified by the temporal arguments of their predicates which subsumes XY-stratified logic programs, and extend it by allowing more powerful logic predicates expressing '>' and '+' primitives needed for greedy algorithms. In Section 4, therefore we show that these extended programs can be expanded into equivalent XY-stratified programs via simple rewritings defined by the arithmetic functions they use. While this rewriting defines the formal semantics of our greedy programs, the equivalent programs produced by the rewriting would be very inefficient if implemented with the standard approach used for XY-stratified programs in systems such as $\mathcal{LDL}^{++}$ (Arni et al. 2003) and DeALS (Shkapsky et al. 2013; Yang et al. 2015). Therefore, we propose a modification of the Iterated Fixpoint computation that solves this problem and actually achieves asymptotic optimality in the implementation of many greedy algorithms, which are discussed in details in Section 5.

2 Local Stratification and Iterated Fixpoint

Let us now recall the definition of local stratification for Datalog programs where rules have negated goals:

Definition 1. *A program P is locally stratified if it is possible to partition its Herbrand base B_P into a countable number of subsets, called strata, $B_0, B_1, \ldots$, such that for every $r \in ground(P)$ the stratum of the head of r is strictly higher that the strata of its negated goals, and higher or equal to the strata of its positive goals.*

Each locally stratified program has a unique stable model, called its perfect model, whereby its abstract semantics has many desirable properties (Przymusinski 1988). Moreover, once the aforementioned stratification $B_0, B_1, \ldots$ is known for a program P, then the *Iterated Fixpoint* procedure can be used to compute the perfect model of P. The iterated fixpoint can be defined using the *immediate-consequence* operator for the rules in $ground(P)$ whose head is in stratum B_K: let T_K denote the immediate consequence operator for instantiated rules belonging to the K-th stratum. and let $\mathbf{T}_K$ denote he inflationary version to this operator defined as $\mathbf{T}_K(I) = T_K(I) \cup I$.

Then the *iterated computation* on our locally stratified program P is performed by starting from $M_0 = \mathbf{T}_0^{\uparrow\omega}(\emptyset)$ and then continuing with $M_{K+1} = \mathbf{T}_K^{\uparrow\omega}(M_K)$.

While the iterated fixpoint procedure seems to provide an efficient operational semantics for computing of the perfect model for locally stratified program, in reality this is not the case, because of a number of problems, including the fact that the existence of local stratification for a given program represents an undecidable question (Palopoli 1992).

To address these problems we will introduce the notion of programs that are locally stratified by the positive numbers that appear in a distinguished argument of their predicates, that we call temporal argument. Thus, we propose simple syntactic conditions that assures that (i) a local stratification exists and (ii) the actual strata are identified quite easily. We then turn to the issue of improving the efficiency of the iterated fixpoint procedure, by basically skipping over the computation of strata that do not produce any useful result. We will thus identify simple syntactic conditions that make this optimization possible, and we will show that greedy algorithms are naturally expressed under this restricted syntax, producing declarative Datalog programs that preserve the desirable complexity properties of their procedural counterparts.

3 Temporally Stratified Datalog$_{1S}$ Programs

Let us consider Datalog programs where the first argument is a non-negative integer represented by the successor notation: $0, s(0), s(1), \ldots, s^n(0)$ described in (Chomicki and Imielinski 1988). These are known as Datalog$_{\mathtt{1S}}$ programs, and have been studied extensively in (Chomicki 1990), where the authors called the `1S` argument the *temporal argument*, a naming convention that we will also follow in this paper. For example consider the following program:

Example 1 (*A Datalog$_{\mathtt{1S}}$ program defining all even positive integers.*)

$$\begin{array}{ll}\mathtt{int(even, 0).} & \\ \mathtt{int(even, s(J))} \leftarrow & \neg\mathtt{int(even, J).}\end{array}$$

Thus the temporal argument in our `int` predicate is the last one, where a positive integer n is represented by n applications of the function symbol s to zero. We will use the short-hand $s^n(X)$ to denote the application of s to X repeated n times $s(s(\ldots s(X)\ldots))$.

Thus the Herbrand universe for the above program is $\{\mathtt{s^n(0)}, \mathtt{s^n(even)}\}$ where n denotes an arbitrary non-negative integer (under the convention that $\mathtt{s^0(X)} = \mathtt{X}$, and thus $\mathtt{s^0(even)} = \mathtt{even}$).

Now, let P be a Datalog$_{\mathtt{1S}}$ program, then the *temporal layering of P* is the one obtained by assigning each atom in its Herbrand Base B_P to the layer n whenever the last argument of the atom is $\mathtt{s^n(c)}$, with `c` an arbitrary constant. For instance the temporal layering of the above program in Example 1 is as follows:

Layer

0: $\texttt{int(s}^{\texttt{n}}\texttt{(0),0)}$, $\texttt{int(s}^{\texttt{n}}\texttt{(even),0)}$, $\texttt{int(s}^{\texttt{n}}\texttt{(0),even)}$, $\texttt{int(s}^{\texttt{n}}\texttt{(even),even)}$

1: $\texttt{int(s}^{\texttt{n}}\texttt{(0),s(0))}$, $\texttt{int(s}^{\texttt{n}}\texttt{(even),s(0))}$, $\texttt{int(s}^{\texttt{n}}\texttt{(0),s(even))}$, $\texttt{int(s}^{\texttt{n}}\texttt{(even),s(even))}$

...

k: $\texttt{int(s}^{\texttt{n}}\texttt{(0),s}^{\texttt{k}}\texttt{(0))}$, $\texttt{int(s}^{\texttt{n}}\texttt{(even),s}^{\texttt{k}}\texttt{(0))}$, $\texttt{int(s}^{\texttt{n}}\texttt{(0),s}^{\texttt{k}}\texttt{(even))}$, $\texttt{int(s}^{\texttt{n}}\texttt{(even),s}^{\texttt{k}}\texttt{(even))}$

We will focus on programs that are locally stratified according to their temporal layering:

Definition 2 *A Datalog$_{1S}$ program P will be said to be* temporally stratified *if P is locally stratified according to its temporal layering.*

Thus the program in Example 1 is temporally stratified. However, the program in Example 2, below, it is not locally stratified, although it has the same Herbrand base, and can be assigned the same temporal layering as Example 1:

Example 2 (*A Temporally Layered Program that is not locally stratified*)

```
int(even, 0).
int(even, J) ←  ¬int(even, s(J)).
```

The simple programs so far considered only use one predicate, but to express powerful algorithms we need to consider programs featuring several predicates within each given layer. For these programs, deciding whether they are locally stratified, and determining the perfect model for those that are stratified, can be quite challenging in general. A solution is however at hand for the large class of such problems that satisfies the notion of XY-stratification discussed next.

4 XY-Stratified Programs

Temporally stratified programs have been explored in the past. In particular, (Zaniolo et al. 1993) introduced XY-stratified programs that are efficiently supported in $\mathcal{LDL}^{++}$ and DeALS (Shkapsky et al. 2013), (Yang et al. 2015) and were also used in a number of advanced applications (Guzzo and Saccà 2005), (Borkar et al. 2012). A generalized version of XY-stratification, called explicitly stratified logic programs (Lausen et al. 1998), was then used to model active rules and other interesting applications. Take for instance the transitive closure prgram for a graph:

Example 3 (*Transitive Closure expressed in Datalog*)

```
cl(X, Z) ←  arc(X, Z).
cl(X, Z) ←  cl(X, Y), arc(Y, Z).
```

The differential fixpoint (a.k.a. seminaive fixpoint) of this program can be expressed by the following program, where `dcl` is the delta version of `cl`.

Example 4 (*Differential rules used in computing the transitive closure of* `arc`)

```
dcl(X, Z, 0) ←   arc(X, Z).
dcl(X, Z, J1) ←  dcl(X, Y, J), arc(Y, Z), J1 = J+1, ¬previous(X, Z, J).

previous(X, Z, J1) ←  dcl(X, Z, J), J < J1.
```

The above program is a Datalog_{1S} program expressed by a slightly different notation (Chomicki 1990). In fact, instead of representing the successor of integer `J` by `s(J)`, we represent it here by `J+1` where `+1` is a postfix function symbol. Therefore, we can easily conclude that the program above is temporally layered by the last argument (i.e., `J` and `J1`) in our predicates. However we cannot conclude that the resulting program is locally stratified, because the definition of $ground(P)$ does not prevent us from instantiating `J < J1` to values where `J` is actually larger than `J1`. This problem can be solved by a simple rewriting of the rules to explicitly define $>$ using the past values of `dcl` kept in lower strata, which we will write as h`dcl` (for historical `dcl`).

Example 5 (*Differential rules used in computing the transitive closure of* `arc`)

$$\begin{array}{ll}
\texttt{dcl(X,Z,0)} \leftarrow & \texttt{arc(X,Z)}.\\
\texttt{dcl(X,Z,J1)} \leftarrow & \texttt{dcl(X,Y,J)},\texttt{arc(Y,Z)},\texttt{J1} = \texttt{J+1},\neg h\texttt{dcl(X,Z,J)}.\\
h\texttt{dcl(X,Z,J)} \leftarrow & \texttt{dcl(X,Z,J)}.\\
h\texttt{dcl(X,Z,J1)} \leftarrow & h\texttt{dcl(X,Z,J)},\texttt{J1} = \texttt{J+1}
\end{array}$$

The resulting program is temporally stratified, i.e., locally stratified by the temporal layering established by the last argument of its recursive predicates. Observe that in the program above we only have two kinds of temporal arguments: `J` and `J+1` = `J1`, i.e., a variable and its immediate successors. For these programs there is a simple test that allows us to determine if they are locally stratified. This is the XY-stratification test that is performed by renaming the predicates in the recursive rules that have a temporal argument, as follows: in each rule r rename with the sufix '`_old`' the goals having as temporal argument `J` when the temporal argument in the head of r is `J1` = `J+1`. The program so obtained is called the *bi-state* version of the original program. Then, a program P is said to be XY-stratified when its bi-state version is stratified. XY-stratified programs are locally stratified and their perfect model can be efficiently computed using their bi-state version (Zaniolo et al. 1993). For instance, the bi-state version of the program in Example 5 is as follows:

Example 6 (*The Bistate Version of Example 5*)

$$\begin{array}{ll}
\texttt{dcl(X,Z,0)} \leftarrow & \texttt{arc(X,Z)}.\\
\texttt{dcl(X,Z,J1)} \leftarrow & \texttt{dcl(X,Y,J)},\texttt{arc(Y,Z)},\texttt{J1} = \texttt{J+1},\neg h\texttt{dcl\_old(X,Z,J)}.\\
h\texttt{dcl(X,Z,J)} \leftarrow & \texttt{dcl(X,Z,J)}.\\
h\texttt{dcl(X,Z,J1)} \leftarrow & h\texttt{dcl\_old(X,Z,J)},\texttt{J1} = \texttt{J+1}.
\end{array}$$

We have obtained a program that is stratified (e.g., with the following strata:`1:{arc}, 2:{dcl_old,` h`dcl_old}, 3:{dcl}, 4:{`h`dcl}`).

Therefore, XY-stratification provides a simple test to verify that temporally layered programs are locally stratified and thus temporally stratified. As proven in (Zaniolo et al. 1993), the perfect model of these programs can be computed as follows (for clarity we refer to predicates without the suffix '`_old`' as 'new'):

Perfect Model Computation for XY-stratified Programs:

(i) use the bistate program to derive the values for the new predicates, and
(ii) re-initializing the values of the '`_old`' predicates with those of the predicates just computed, and then the values of the 'new' predicates.

These steps repeated until (i) stops producing new tuples, construct the perfect model for our XY-stratified (and therefore temporally stratified) program. This basic procedure delivers good performance on many simple problems including the seminaive computation of the least fixpoint of Example 2, above, but a more sophisticated approach is needed to achieve optimal performance for logic programs expressing greedy algorithms, since these are considerably more complex.

To simplify the expression, and also the compilation, of greedy algorithm, we will introduce the notation `not(...)`. Thus, Example 4 can be re-expressed as follows:

Example 7 (*Example 4 re-expressed using* `not( )`)

```
dcl(X, Z, 0) ←      arc(X, Z).
dtrcl(X, Z, J1) ←   dcl(X, Y, J), arc(Y, Z), J1 = J+1,
                    not(dcl(X, Z, K), K < J).
```

In general, a program with a goal '`not(condition)`' should be viewed as the shorthand of the program derived by (i) replacing '`not(condition)`' with ¬`newp(SVlist)`, (where `SVlist` denotes the variables shared between `condition` and the rest of the rule), and (ii) adding the rule: `newp(SVlist)` ← `condition`.

5 Greedy Algorithms

Suppose now that `warc(X, Z, W)` describes a directed graph where `W` is the positive weight of the arc from `X` to `Z`. For now, let us assume that all such weights are integers. Then, a greedy algorithm to find the shortest path between node pairs is as follows:

Example 8 (*A greedy algorithm to find shortest paths between node pairs*)

```
wtc(X, Z, W) ←     warc(X, Z, W).
wtc(X, Z, Cz) ←    wtc(X, Y, Cy), not(wtc(X, _, C), C < Cy),
                   warc(Y, Z, W), Cz = Cy + W.
```

Thus in the recursive rule, we use the `not` construct to find the shortest distance `Cy` from a node `X` to a node `Y` and, for each arc in `warc(Y, Z, W)`, we add a path from `X` to `Z` of length `Cz = Cy + W`. Our objective is to re-express this as an equivalent XY-stratified program. In order to do that, we will re-write the program into the following one, where we re-express '+' using the successor logic of Datalog$_{1S}$.

Example 9 (*A temporally stratified program to find shortest paths between node pairs*)

```
wtc(X, Z, W) ←             warc(X, Z, W).
succadd(X, Y, W, C1) ←     wtc(X, Y, Cy), not(wtc(X, _, C), C < Cy),
                           warc(Y, Z, W1), W = W1 + 1, C1 = Cy + 1.
succadd(X, Y, W, C1) ←     succadd(X, Y, W + 1, C), C1 = C + 1.
wtc(X, Z, Cz) ←            succadd(X, Y, 0, Cz).
```

Thus recursive `succadd` rule expresses '+' by raising by 1 the value of `Cy` while replacing `W + 1` with `W` until this becomes zero: at this point the addition has been completed, whereby `Cz` is the distance of `Z` from `X`. We can now expand the `not(...)` goal, and finally verify that the resulting program is XY-stratified, by the first goals in the second and third rule as `wtc_old` and `succadd_old`, respectively. For the rules resulting from the expansion of `not`(...) we proceed as in Example 4. Then we obtain a bistate program which is stratified: therefore the original program in Example 9 is XY-stratified and thus temporally stratified.

A program such as that in Example 9, where the rewriting of its $<, +$, and `not` goals produce a temporally stratified programs will be called an *Implicit Temporally Stratified* (ITS) program. Now, many programs expressing greedy algorithms can be transformed into XY-stratified programs that provide a formal semantics for such programs, since these are known to be locally stratified. The perfect model for these programs can also be computed using the standard bistate based computation of XY-stratified programs, but as discussed next, this computation would fail to deliver optimal performance for the program in Example 9 and other greedy programs.

The obvious problem with the standard bistate-based computation of the program in Example 9 is that in order to derive $\mathtt{Cz} = \mathtt{Cy} + \mathtt{W}$, we go through the computation of the $\mathtt{W} - \mathtt{1}$ temporal strata that take us from `Cy` to `Cz`, even though no new `wtc` value might be produced in step (i) and in step (ii) of the Perfect Model Computation for XY-stratified programs discussed on page 5. In order to bypass this sequence, we might consider jumping directly to stratum with temporal argument `Cz`, but that might not be correct, since the same rule that has now produced `Cz` might have previously produced a value $\mathtt{C'z}$, $\mathtt{Cy} < \mathtt{C'z} < \mathtt{Cz}$, which must be considered before `Cz`. The solution to this problem is obvious: (1) we store the value `Cz` produced by the second rule into a *priority queue* (PQ) and then (2) we fetch (and remove) the least value from PQ, and use it as the next value of the temporal argument. Needless to say, our PQ is exactly the data structure used in Dijkstra's shortest path algorithm and other greedy algorithms. Thus we can achieve an optimal computation of our greedy algorithms by simply replacing the +1 successor operation with a PQ store+fetch operation.

This PQ optimization however it is is not applicable to all temporally stratified programs and in particular to Example 1, due to the fact that the only goal in its rules is a negated goal. To avoid this potential problem we now introduce the notion of *Strict Implicit Temporally Stratified* (SITS) that assures the applicability of the PQ optimization.

Definition 3 *An ITS program will be said to be* strict *when every rule containing negated goals also contains some positive goal which has a temporal argument that is $\geq$ than the temporal argument of every negated goal.*

Thus this condition excludes the program in Example 1, and also disallows the following rule that satisfies the standard notion of implicit temporal stratification:

$$\begin{aligned}\mathtt{wtc(X,Z,Cz)} \leftarrow \quad & \mathtt{wtc(X,Y,Cy), arc(Y,Z,W), Cz = Cy + W,}\\ & \mathtt{not(wtc(X,\_,C), C < Cz).}\end{aligned}$$

The problem with this rule is that it offers no assurance that $\mathtt{Cy} \geq \mathtt{C}$. A rule like the one above is no problem for the iterated fixpoint procedure that visits every successive value of the temporal argument, but it cannot be supported in a computation that jumps from the current temporal argument to the next temporal value extracted from PQ. However teh strictness condition solves this problem and maket it possible to use the following computation:

Algorithm 1 Computation of Perfect Model for SITS Programs

```
1: Initialize the priority queue (PQ) to empty.
2: Let $M := \mathbf{T}_0^{\uparrow\omega}(\emptyset)$
3: Add the values of temporal arguments generated in step 2 to PQ.
4: Repeat the following three steps until PQ becomes empty:
    5: Remove the least temporal argument $K$ from (PQ) and
    6: Let $M := \mathbf{T}_K^{\uparrow\omega}(M)$.
    7: Add the values of the newly generated temporal arguments to PQ.
```

Thefore, SITS programs can be computed efficiently by a simple optimization of the iterated fixpoint algorithm that consists of skipping over unproductive values of temporal arguments:

5.1 Beyond Integers

In our discussion so far, we have assumed that temporal arguments are positive integers, but our treatment of greedy algorithms generalizes to the case in which we have arbitrary positive numbers, not just integers, whereby arbitrary positive weights can, e.g., be used as arc weights in our graphs. This conclusion follows from the argument presented in (Mazuran et al. 2013) where it was observed that non-integers could be represented as rational numbers sharing a common very large denominator D whereby all computations can be emulated by integer arithmetics on their numerators. Now the standard mantissa+exponent internal representation of real and floating-point numbers, that is used in modern hardware/firmware, does exactly that—modulo some round-off. For instance, for a decimal floating point. the smallest value of exponent supported might be -95 (or smaller), whereby every number can be viewed as the numerator over the denominator $D = 10^{95}$. (For simplicity, we have used a decimal base, but the same conclusions hold for other bases.) Naturally, precision is limited by the fact that the mantissa is of finite length, and thus, e.g., the operation of addition becomes a rounded-off addition. Rounded-off addition can also be easily expressed in Datalog$_{1S}$ whereby the expanded resulting program is still XY-stratified, and the overall formal semantics remains valid. Of course, round-off is also a concern at the operational semantics level, where it can be addressed by the use of double precision and other techniques used when algorithms are expressed in procedural languages. Once he/she selects single or double precision, our user is assured an efficient excution for greedy algorithms owing to the fact that the implementation will not step through each successive floating point number, but jumps directly the next number in the PQ.

Using floating-point numbers and real arithmetic, we can now express a cornucopia of greedy algorithms, starting with the single-source Dijkstra algorithm shown below.

Example 10 (*Single source Dijistra's Algorithm*)

$$\begin{array}{ll}
\mathtt{wtc(X,W)} \leftarrow & \mathtt{warc(a,W).} \\
\mathtt{wtc(Z,Cz)} \leftarrow & \mathtt{wtc(Y,Cy),not(wtc(Y,C),C < Cy),} \\
& \mathtt{warc(Y,Z,W),Cz = Cy + W.}
\end{array}$$

Since, efficient Datalog implementations, such as DeALS (Shkapsky et al. 2013), use Hashing and other indexing techniques to achieve a constant-time computation of the recursive rule above for each value of Y, an optimal performance can be expected for the Dijistra's algorithm above, and similar observations can be made for the other greedy algorithms which which require not special data structure other than PQ. In particular this is true for the Traveling Salesman's Program (TSP) discussed next, which closely emulates its procedural counterpart, thus achieving optimal asymptotic complexity.

Traveling Salesman's Greedy Heuristics

Given an undirected graph, g(X, Y, Cxy), the exit rule selects an arbitrary node X, from which to start the search. Then, the second rule selects candidate new nodes, using the conditions Y<>a, and not(tspath(_, Y, C1), C1 < C) ensures that we do not cycle back to the initial node ad previously derived nodes.

Finally the third rule selects from the candidate new nodes cand(Y, C) the one that has the shortest distance from a.

Example 11 (*Travelling Salesman's starting at node* a)

$$\begin{array}{ll}
\mathtt{tspath(a,0)} \leftarrow & \mathtt{node(a).} \\
\mathtt{cand(Y,C)} \leftarrow & \mathtt{tspath(X,Cx),g(X,Y,Cxy),Y{<}{>}a,} \\
& \mathtt{not(tspath(Y,C1),C1 < C),C = Cx + Cxy.} \\
\mathtt{tspath(Y,C)} \leftarrow & \mathtt{cand(Y,C),not(cand(\_,C1),C1 \leq C).}
\end{array}$$

Thus, this algorithm will work correctly under the assumption that there are no ties, i.e., no two arcs departing from the same node have the same weight. In the situation where there are ties, we can employ a construct such as choice (Greco et al. 1992), which models don't care non-deterministic semantics via a special class of stable models called choice models. Alternatively, we can fall back on the solutions used by procedural programmers. For instance, since nodes are represented by natural numbers, or by elements of an ordered domain, we can expand the third rule in Example 11 as follows:

Example 12 (*Extrema as tie-breaker: alternative for 3^{rd} rule in Example 11*)

$$\mathtt{tspath(min\langle Y\rangle,C)} \leftarrow \quad \mathtt{cand(Y,C),not(cand(\_,C1),C1 \leq C).}$$

This program is SITS once we assume that the min aggregate is defined as follows:

$$\begin{array}{ll}
\mathtt{mtspath(Y,C)} \leftarrow & \mathtt{cand(Y,C),not(cand(\_,C1),C1 < C).} \\
\mathtt{tspath(Y,C)} \leftarrow & \mathtt{mtspath(Y,C),not(mtspath(Y,C1),C1 < C).}
\end{array}$$

Here the intra-layer stratification uses `cand` al the first level, and `mtspath` at the second level, and `tspath` at the top level. Thus, while the body of the second rule eliminates the nodes having a smaller C, the aggregate in the head only retains the first (i.e., the smallest) `Y` out of those candidate nodes that share the same `C`.

While the use of min or max aggregates could be all a user wants in most practical applications, from a conceptual viewpoint we might regret the fact that we have given up non-determinism, and we can only generate one TSP path rather than a different one at each run. However, non-determinism can be recovered by a builtin predicate, such a hash function `h(_)`, that reorders its input nondeterministically. Then our program becomes:

Example 13 (*Using randomized hashing for non-determinstic TSP*)

```
tspath(a, 0) ←  node(a).
cand(Y, C) ←    tspath(X, Cx), g(X, Y, Cxy),
                Y<>a, not(tspath(Y, C1), C1 < C), C = Cx + Cxy.
slct(SL, C) ←   cand(Y, C), not(cand(_, C1), C1 ≤ C),
tspath(L, C) ←  slct(L, C), not(slct(L1, C), h(L) > h(L1)).
```

Here the the intra-layer stratification has `cand` below `slct` which is below `tspath`. Similar techniques for breaking ties can be used in Prim's and other algorithms.

Prim's algorithm

We build a tree with nodes `st(X, Cx)` where `C` is a node and `Cx` is the cost of the tree when `X` was produced. We start from a node a with cost 0. Then for the current level C, we find all the new nodes reachable from this or previous nodes. This provides a set of candidates for which, in the third rule, we take the one that delivers the least cost.

Example 14 (*Prim's minimum cost spanning tree.*)

```
st(a, 0).
cand(C1, Y) ←  st(C, _), st(Cx, X), Cx ≤ C,
               arc(X, Y, Cxy), Y <> a,
               not(st(Cy, Y), Cy < C), C1 = C + Cxy.
st(C1, Y) ←    cand(C1, Y), not(cand(C, _), C < C1).
```

Our example assumes that no two arcs have the same weight. When this is not the case, we will use the same tie-breaking solutions used for TSP.

Huffman Encoding

Huffman coding is a lossless data compression algorithm. The idea is to assign variable-length codes to input characters, with lengths of the assigned codes based on the frequencies of corresponding characters. Frequent characters are assigned shorter codes. The variable-length prefix codes are bit sequences assigned to input characters in such a way that no code of a character is the prefix of the code of another character.

The input is the list of unique characters along with their frequency of occurrences and output is the Huffman tree. The basic algorithm is as follows. We start from a set of facts, `token(Char, Freq)` which corresponds to the leaf nodes of the Huffman tree. Then the algorithm can be expressed as follows:

Example 15 (*Huffman Encoding Algorithm*)

```
huf(F, X, 0, 0) ←          token(X, F).
huf(H, nil, H1, H2) ←
        huf(H1, _, _, _), not(huf(H11, _, _, _), H11 < H1),
        huf(H2, _, _, _), H1 < H2, not(huf(H22, _, _, _), H22 < H2),
        H22<>H1,  H = H1 + H2.
```

For instance, say that we have three facts: `token(a, 4). token(b, 5). token (c,10).` Then the first step consists in executing the rules whose body layer is 0: these are the exit rules, since their bodies consists of facts, which are always viewed as belonging to zero layer. This produces the following leaf nodes in our tree (identified by the fact that their left and right subtrees are both 0).

`huf(4, a, 0, 0).`   `huf(5, b, 0, 0).` `huf(10, c, 0, 0)`

Also as a result of this step, we have that the values 4, 5, and 10 are entered into the priority PQ. Now, the system takes the least of these values and evaluates the rules for that layer. The rules produce nothing at layer 4, so we move to next layer, 5 where the second rule produces:

`huf(9, nil, 4, 5)`

Thus the system has extracted two nodes with the minimum frequency from the min heap and generated a new node whose weight is the sum of those two.

At this point we have only 9 in the PQ, and where no node at level below 9 is still free the evaluation of the second rule produces nothing, but removes 9 from the PQ. The next value in the PQ is thus 10, and the evaluation of our rule at level 10 produces:

`huf(19, nil, 9, 10)`

At this point, the evaluation at layer 19 produces no new value, whereby the PQ becomes empty, and the computation terminates.

Kruskal's Algorithm

Kruskal's algorithm also constructs a minimum spanning tree for a connected weighted unordered graph. Thus an edge of a graph is represented by a fact `edge(A, B, W)` where `A` $<$ `B`. At each step, the algorithm selects a least-cost edge among those that do not connect previously connected nodes. Thus, in the example below, the first two rules state that each node is connected to itself, starting at level 0. Then, say that at level C we add the new edge `tree(X, Y, C)`, connecting two nodes which, until level `C` were still disconnected.

Then, the last rule is executed that determines all the nodes `X1` and `Y1` respectively connected with `X` and `Y`. Thus, we define

```
mM(X, Y, X, Y) ←   X < Y.
mM(X, Y, Y, X) ←   X > Y.
```

then we see that `mM(X1, Y1, S, L)` it returns `S` and `L` as respectively the smaller and larger of these two. Then we connect to `S` all the nodes previously connected to `L`, i.e., the `Ln` nodes in the last rule.

Example 16 (*Kruskal's Algorithm*)

```
connt(X, X, 0) ←     edge(X, Y).
connt(Y, Y, 0) ←     edge(X, Y).
tree(X, Y, C) ←      tree(_, _, C), edge(X, Y, Cxy), not(connt(X, Y, C1), C1 < C),
                     C = Clast + Cxy.
connt(S, Ln, C) ←    tree(X, Y, C), connt(X1, X, C), connt(Y, Y1, C),
                     mM(X1, Y1, S, L), connt(L, Ln, C).
```

Unlike our previous algorithms, the performance of Kruskal's under our formulation cannot be guaranteed to be optimal, since connectivity is not supported by the special union-find data structure.

6 Conclusion

The non-monotonic constructs proposed in this paper introduce a simple declarative extension for deductive databases that greatly enhances their effectiveness in a range of applications and thus achieves the same optimal time complexity of procedural code algorithms—assuming that these do not make use of special data structures such as Union-Find used by Kruskal's minimum spanning tree algorithm. In fact, we have shown that the greedy optimizations of procedural algorithms follow directly from the need to achieve an efficient implementation for the iterated fixpoint procedure of locally stratified programs. From a theoretical viewpoint, this reveals the computational upside of non-monotonic semantics classes that in the past were primarily analyzed for their intractability downside. From a practical viewpoint, these results allow us to express and implement efficiently in Datalog systems significant algorithms expressed in declarative logic, while achieving the same asymptotic complexity as their procedural counterparts. Indeed, support for strict local stratification can be easily achieved through extensions of XY-stratification, which is now part of DeALS (Shkapsky et al. 2013),(Yang et al. 2015). The integrated support of monotonic aggregates in recursion, greedy algorithms, and XY-stratification supported by our system entails a declarative expression and efficient implementation of complex algorithms, and provides further evidence that this is indeed an age of renaissance for Datalog and deductive databases.

Acknowledgements

The author would like to thank Alex Shkapsky, Mohan Yang, and the referees for many suggested improvements.

References

Abiteboul, S., Bienvenu, M., Galland, A., and Antoine, E. 2011. A rule-based language for web data management. In *PODS*. 293–304.

Afrati, F. N., Borkar, V. R., Carey, M. J., Polyzotis, N., and Ullman, J. D. 2011. Map-reduce extensions and recursive queries. In *EDBT*. 1–8.

Arni, F., Ong, K., Tsur, S., Wang, H., and Zaniolo, C. 2003. The deductive database system ldl++. *TPLP 3,* 1, 61–94.

Borkar, V. R., Bu, Y., Carey, M. J., Rosen, J., Polyzotis, N., Condie, T., Weimer, M., and Ramakrishnan, R. 2012. Declarative systems for large-scale machine learning. *IEEE Data Eng. Bull. 35,* 2, 24–32.

Chomicki, J. 1990. Polynomial time query processing in temporal deductive databases. In *Proceedings of the Ninth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*. PODS '90. 379–391.

Chomicki, J. and Imielinski, T. 1988. Temporal deductive databases and infinite objects. In *PODS*. 61–73.

Gottlob, G., Orsi, G., and Pieris, A. 2011. Ontological queries: Rewriting and optimization. In *ICDE*. 2–13.

Greco, S. and Zaniolo, C. 1998. Greedy algorithms in datalog with choice and negation. 294–309.

Greco, S. and Zaniolo, C. 2001a. Greedy algorithms in datalog. *TPLP 1,* 4, 381–407.

Greco, S. and Zaniolo, C. 2001b. Greedy algorithms in datalog. *TPLP 1,* 4, 381–407.

Greco, S., Zaniolo, C., and Ganguly, S. 1992. Greedy by choice. In *PODS*. 105–113.

Guzzo, A. and Saccà, D. 2005. Semi-inflationary DATALOG: A declarative database language with procedural features. *AI Commun. 18,* 2, 79–92.

Hellerstein, J. M. 2010. Datalog redux: experience and conjecture. In *PODS*. 1–2.

Kolaitis, P. G. 1991. The expressive power of stratified logic programs. *Inf. Comput. 90*, 50–66.

Lausen, G., Ludäscher, B., and May, W. 1998. On active deductive databases: The statelog approach. In *Transactions and Change in Logic Databases*. 69–106.

Mazuran, M., Serra, E., and Zaniolo, C. 2013. A declarative extension of horn clauses, and its significance for datalog and its applications. *TPLP 13,* 4-5, 609–623.

Mumick, I. S., Pirahesh, H., and Ramakrishnan, R. 1990. The magic of duplicates and aggregates. In *VLDB*. 264–277.

Mumick, I. S. and Shmueli, O. 1995. How expressive is stratified aggregation? *Annals of Mathematics and Artificial Intelligence 15*, 407–435.

Palopoli, L. 1992. Testing logic programs for local stratification. *Theor. Comput. Sci. 103,* 2, 205–234.

Przymusinski, T. C. 1988. Perfect model semantics. In *ICLP 1988*.

Ross, K. A. and Sagiv, Y. 1997. Monotonic aggregation in deductive database. *J. Comput. Syst. Sci. 54,* 1, 79–97.

Shkapsky, A., Zeng, K., and Zaniolo, C. 2013. Graph queries in a next-generation datalog system. *PVLDB 6,* 12, 1258–1261.

Yang, M., Shkapsky, A., and Zaniolo, C. 2015. Parallel bottom-up evaluation of logic programs: Deals on shared-memory multicore machines. In *ICLP 2015, Cork Ireland.*

Zaniolo, C. 2011. The logic of query languages for data streams. In *Logic and Databases 2011. EDBT 2011 Workshops*. 1–2.

Zaniolo, C., Arni, N., and Ong, K. 1993. Negation and aggregates in recursive rules: the ldl++ approach. In *DOOD*. 204–221.

Zaniolo, C., Ceri, S., Faloutsos, C., Snodgrass, R. T., Subrahmanian, V. S., and Zicari, R. 1997. *Advanced Database Systems.* Morgan Kaufmann.