Greedy by Choice

Sergio Greco* Università della Calabria 87030 Rende, Italy sergio@mcc.com

Carlo Zaniolo University of California at Los Angeles LosAngeles, CA 90024 zaniolo@cs.ucla.edu

Sumit Ganguly University of Texas at Austin Austin, TX 78712 sumit@cs.utexas.edu

Abstract

The greedy paradigm of algorithm design is a well known tool used for efficiently solving many classical computational problems within the framework of procedural languages. However, it is very difficult to express these algorithms within the declarative framework of logic-based languages. In this paper, we extend the framework of Datalog-like languages to provide simple and declarative formulations of such problems, with computational complexities comparable to those of procedural formulations. This is achieved through the use of constructs, such as `least` *and* `choice`*, that have semantics reducible to that of negative programs under stable model semantics. Therefore, we show that the formulation of greedy algorithms using these constructs lead to a syntactic class of programs, called stage-stratified programs, that are easily recognized at compile time. The fixpoint-based implementation of these recursive programs is very efficient and, combined with suitable storage structures, yields asymptotic complexities comparable to those obtained using procedural languages.*

1 Introduction

Current research in deductive databases is tackling the formidable problem of providing declarative logic-based semantics and efficient implementations for non-stratified logic programs containing non-monotonic constructs such as negation and aggregates. The interesting theoretical challenges posed by this problem are made more urgent by the fact that these constructs are critically needed to express a host of practical applications, ranging from the 'Bill of Materials' to graph-computation algorithms. Proposed solutions must provide suitable language constructs, declarative non-monotonic semantics and the enabling technology for their efficient implementation—including compilation methods, execution algorithms, and supporting storage structures. The problem of finding efficient implementations for declarative logic-based languages, represents one of the most ardous and lasting research challenges in computer science, and one to which deductive databases have recently offered new ideas and inroads toward solutions. In particular, in this paper, we show that a careful choice of metalevel constructs having a first order semantics can be used to produce declarative formulation and nearly optimal execution for the important class of problems solvable using greedy algorithms.

In Section 2 of this paper, we define extrema and choice-based constructs using meta-level notation and first-order semantics. In Section 3, we show that a combination of these constructs defines programs that are locally stratified according to stage variables—yielding a stable-model semantics, for which a fixpoint characterization is given in Section 4. Thus, in Section 5, we express various greedy algorithms and, in Section 6, we show that, through seminaive refinements and suitable storage structures, a fixpoint computation yields low asymptotic complexity on these problems. In the conclusion we review the many opportunities for further research, including the use of matroid theory to push extrema predicates into choice-based programs.

2 Non-Monotonic Constructs

$\mathcal{LDL}$'s `choice` construct provides an example of the benefits of using meta-level predicates with first-order semantics to extend the power of a Horn-clause language in a way that is easy for users to understand and simple for the system designer to implement. The `choice` construct was introduced in [7], to express non-determinism in a declarative fashion. Thus, a meta-level predicate `choice(X, Y)` is used to denote that the functional dependency $X \rightarrow Y$ must hold in the model

*Work done while visiting MCC. Work partially supported by the project "Sistemi Informatici e Calcolo Parallelo" obiettivo "Logidata+" of C.N.R. Italy.

11th Principles of Database Systems/6/92/San Diego, CA
© 1992 ACM 0-89791-520-8/92/0006/0105...$1.50

defining the meaning of this program.

Example 1: One student per course and one course per student.

```
a_st(St, Crs) ← takes(St, Crs),
                choice(Crs, St), choice(St, Crs).
```

The choice goals in this rule specify that the *a_st* predicate symbol must associate exactly one student to each course, and vice-versa. Thus the functional dependencies $Crs \rightarrow St$, $St \rightarrow Crs$ hold in the (choice model defining the) meaning of a program containing such a rule.

Thus the program of Example 1, with the following set, X, of facts:

```
takes(andy,engl,4).    takes(mark,engl,2).
takes(ann, math,3).    takes(mark,math,2).
```

has the following three choice models:

M_1 = { *a_st(andy, engl, 4), a_st(ann, math, 3)* } ∪ X,
M_2 = { *a_st(mark, engl, 2), a_st(ann, math, 3)* } ∪ X,
M_3 = { *a_st(andy, engl, 4), a_st(mark, math, 2)* } ∪ X. □

Therefore, the meaning of a choice program according to [4] is defined by extracting a maximal subset that obeys the given functional dependencies from the model of the program obtained by stripping out the choice goals. One such model always exists—frequently, several such models could exist denoting that a non-deterministic choice can be made among them. A second milestone in the of choice, was its characterization in terms of negation [9] under stable model semantics [3]. According to [9], choice can be viewed as a meta-level construct with first-order semantics: a rule with choice can be viewed as a short-hand of convenience for a program with negation. The equivalent of the choice rule of Example 1 is:

Example 2:

```
a_st(St, Crs, G) ← takes(St, Crs, G), chosen(Crs, St).
chosen(Crs, St) ← takes(St, Crs, G),
                  ¬diffChoice(Crs, St).
diffChoice(Crs, St) ← chosen(Crs, St̄), St ≠ St̄.
diffChoice(Crs, St) ← chosen(Crs̄, St), Crs ≠ Crs̄.
```

□

Negative programs so constructed have at least one total stable model (frequently, several stable models, denoting the presence of non-determinism) [9]. [1] The characterization of non-determism by means of negation, provided in [9] unifies two important notions of logic programming and sheds new insights on the nature of stable models. Furthermore, it leads to a constructive semantics where a simple and efficient implementation of the construct is obtained by the memoing of previous results [4]. In a nutshell, the constructive semantics recognizes that a choice program re-written as the program P above consists of two kinds of rules:

1. the *chosen* rule(s) with head the `chosen` predicates and a negated `diffchoice` goal in the body,
2. the remaining rules, including (a) the original program rules without choice goals, (b) the rewritten positive rule containing the goal `chosen`, and also (c) the auxiliary rules defining `diffChoice`. (For now, we can assume that these are Horn rules.)

If T_C and T', define the immediate consequence operator respectively, for the *chosen* rules (1), and the remaining rules (2), then we can provide a constructive semantics as follows [4].

Let I be an interpretation for the re-written program P. Then, g denotes the non-deterministic operator that maps I into a set that is either empty or contains *exactly* one element, defined as follows:

1. if $T_C(I) - I = \emptyset$ then $g(I) = \emptyset$,
2. if $T_C(I) - I \neq \emptyset$ then $g(I) = \{x\}$, where $x \in T_C(I) - I$.

Thus, applying g to I means computing all the new implied facts and then arbitrarily selecting a member of this set. We will call g, the *one-consequence* operator for P. Now, let $\gamma(I) = g(I) \cup I$ denote the inflationary version of g, and let $Q(I) = T'(I) \cup I$ be the inflationary version of T'. Consider the following transfinite procedure that alternates between the application of the *one-consequence* operator γ (i.e., the firing an instance of a rule *chosen*) and the firing of all the remaining rules till saturation.

Moreover, the choice fixpoint procedure is also (non-deterministically) complete in the following sense: if Q^∞ is finite at each iteration, then, the choice fixpoint algorithm computes every stable model, for suitable instantiations of the non-deterministic one-consequence operator γ.

[1] When the program contains several rules with choice, a distinguished $\texttt{chosen}_i$ and $\texttt{diffChoice}_i$ must be generated for each rule —a different value of i for each choice rule.

Choice Fixpoint:
begin
$S' := \emptyset$;
repeat
$S := S'$;
$S' := Q^{\infty}(\gamma(S))$;
until $S' = S$
end.

Then we have the following result [4]:

Lemma 1 *Let P be a positive program where rules contain choice goals. Then the* Choice Fixpoint Procedure *produces all stable models of P.* □

Furthermore, when there are nofunction symbols (Datalog programs), we have a finite Herbrand base and the following lemma also holds:

Lemma 2 *Let P be a positive Datalog program where rules contain choice goals. Then*

- *The Choice Fixpoint procedure is (non-deterministically) complete for P.*
- *The data complexity of computing a stable model for P is polynomial time.*

□

The different stable models for P, called the *choice models* for P, are produced by different instances of the non-deterministic γ function. An efficient implementation for choice programs only requires memorization of the *chosen* predicates; from these, the *diffChoice* predicates can be generated on-the-fly. Moreover, for Datalog programs, the Choice Fixpoint Procedure terminates in polynomial time (although, computation of stable models is, in general, NP-hard). Thus, choice provides a good example of how, out of the very powerful class of stable models that in the raw is neither suitable for system building or users' consumption, an important subclass can be encapsulated that is amenable to efficient implementation and appealing to intuition.

Meta-level notation is also expedient with extrema aggregates (e.g., min, max, least and most) that can be defined using negation [1]. For instance, to find the names of courses, and, for each course, the students who took the least grade among the students who have grades above 1, we can write the following program:

```
bttm_st(St, Crs, G) ← takes(St, Crs, G), G > 1,
                      least(G, Crs).
```

where the meaning of `least` is defined via the following expansion:

```
bttm_st(S, C, G) ← takes(S, C, G), G > 1,
                   ¬(takes(S', C, G'), G' > 1, G' < G).
```

Thus, the meaning of `least(G,C)` is to select among the bindings that make the body of the rule true, those such that there is no other instantiation of the body of the rule that has the same value of the variable `C` but a lesser value for the variable `G`. Often, `least(G, ())` may be used to select the binding that makes the body of the rule true and has the smallest value of the variable G, without regard to the value of the other variables. This is often abbreviated as `least(G)`.

The use of the metalevel predicates for extrema simplifies the notation and allows the compiler to use a more efficient implementation. However, it does not solve the problem of semantics, since the unrestricted use of, say, `least` in recursion can produce programs that have no accepted declarative meaning (i.e., no total well-founded or stable model exist for such programs [11, 3]). This is different from choice which has efficient implementation and clear semantics even in recursive rules [4]. In this paper, we will provide declarative semantics and efficient implementation for a powerful class of recursive programs featuring an interesting mixture of choice and extrema predicates.

For instance, consider a combination of the previous examples: bi-injective pairs of student/course that have received the lowest grades above 1:

```
bi_st_c(St, Crs, G) ← takes(St, Crs, G), G > 1, least(
                      choice(St, Crs), choice(Crs, St)
```

The meaning of rules containing both `choice` and least is naturally resolved by *applying the rewriting for choice before the rewriting for least* as exhibited by the following intermediate result: [2]

```
bi_st_c(St, Crs, G) ← takes(St, Crs, G), G > 1,
                      least(G), chosen(Crs, St).
chosen(Crs, St) ← takes(St, Crs, G), G > 1,
                  least(G), ¬diffChoice(Crs, St).
diffChoice(Crs, St) ← chosen(Crs, St), St ≠ St.
diffChoice(Crs, St) ← chosen(Crs', St), Crs ≠ Crs'.
```

Therefore, we are selecting bi-injective pairs out of those with bottom grade > 1, and not the bottom grades out of randomly selected bi-injective pairs. (The *least* goal in the top rule can be eliminated since it only recomputes the one in the lower rule). Two stable models for this last rule and the X facts of Example 1 are (omitting *diffChoice* facts):

$M_1 = \{$ *bi_st_c(mark, engl, 2), chosen(mark, engl, 2)* $\} \cup X$
$M_2 = \{$ *bi_st_c(mark, math, 2), chosen(mark, math, 2)* $\} \cup X$

[2] The complete expansion of the second rule, therefore is:
```
chosen(Crs, St) ← takes(St,Crs,G), G>1,
      ¬ diffChoice(Crs,St), ¬(takes(St', Crs',G'),
      G' > 1, ¬ diffChoice(Crs',St'), G' < G).
```

3 Stage Variables

The use of choice in logic programs enables elegant expression of classical graph algorithms and their efficient implementation. For instance suppose that an undirected graph is stored as pairs of edges, `g(Y,X,C)`, `g(X,Y,C)`, with C the cost of the edge. Then a spanning tree for this graph, starting from the source node `a`, can be computed as follows:

Example 3: Spanning tree

```
st(nil, a, 0).
st(X, Y, C) ← st(_, X, _), g(X, Y, C), choice(Y, (X, C)).
```

(If there is only one cost per arc, the variable `C` can be eliminated from `choice`.) □

A different formulation of this problem can be expressed using a stage variable `I` that associates each edge with a unique value of the index `I`, and vice versa. The resulting program is:

```
st(nil, a, 0, 0).
st(X, Y, C, I) ← st(_, _, _, I1), I = I1 + 1,
                 choice(I, (X, Y, C)), choice((X, Y, C), I),
                 g(X, Y, C), choice(Y, (X, C)).
```

The first four goals in the recursive rule are most interesting, inasmuch as they ensure that for each triple (X,Y,C) returned by the remaining two goals, *exactly* one new value is returned for argument `I`. Thus, the argument `I` becomes a *stage variable* that introduces a local stratification in the recursive program. We will thus introduce a new meta-level predicate `next(I)` to simplify the use of stage variables in our programs: `next(I)` operates as a meta-level construct with first-order definition, according to the following macro-expansion:

```
p(W, I) ← next(I), rest_of_body.
```

where W∪{I} is the argument list in the head, is replaced by:

```
p(W, I) ← rest_of_body, p(_, I1), I = I1 + 1,
          choice(I, W), choice(W, I).
```

Thus, our algorithm for finding a spanning tree can be replaced by:

```
st(nil, a, 0, 0).
st(X, Y, C, I) ← next(I), g(X, Y, C), choice(Y, (X, C)).
```

Since `next` is only a short-hand for the use of choice, these program have a declarative semantics based on stable models (the actual program defining this semantics is obtained by applying first the `next` expansion, then the rewriting for `choice` and, finally, the rewriting for `least`). Moreover, as we shall see next, many recursive programs combining `next(I)` and extrema have a stable-model semantics. Take for instance, the computation of minimum spanning trees according to Prim's algorithm:

Example 4: Prim's Algorithm.

```
prm(nil, a, 0, 0).
prm(X, Y, C, I) ← next(I), new_g(X, Y, C, J), J < I,
                  least(C, I), choice(Y, X).
new_g(X, Y, C, J) ← prm(_, X, _, J), g(X, Y, C). □
```

Here the relation `prm` collects arcs in the minimum spanning tree, their cost, and the step at which they were entered. We can easily visualize the computation of our mutually recursive predicate, in terms of a fixpoint procedure: at each new iteration, new arcs `new_g(X,Y,C,I)` connected to current spanning tree are generated (by the last rule). Then, the second rule selects a new arc with least cost of the possible choices; this new arc is added to `prm` while the stage variable is incremented by one. Notice that the choice in `next(I)` forces a new selection for each value of `I`.

4 Semantics

Consider a program P containing rules with negated goals, extrema, next goals, and choice goals, and let Q denote the program obtained from P by,

1. expanding the next goals,
2. removing all choice goals and,
3. rewriting the least goals using negation.

Then, if Q is *locally stratified*, we will say that the original program P is *locally stratified, modulo choice.* Our previous examples are locally stratified modulo choice. Thus, we have the following theorem [2].

Lemma 3 *A stable model exists for every program that is locally stratified modulo choice.* □

Locally stratified programs have a unique stable model (which is also a well-founded model and a perfect model) [8]; but, programs that are stratified modulo choice normally have several stable models. The perfect model of a locally stratified program can be constructed through a possibly infinite sequence of fixpoint computations that saturates stratum j after stratum $j-1$ [8]. A similar procedure, where the operator $Q^{\infty}(\gamma())$ is used to saturate each stratum yields the stable models for programs that are locally stratified modulo choice [10]. In this paper, we concentrate on a subclass of locally stratified programs

modulo choice that we call *stage programs*. These are defined next:

A rule r is said to be a *next rule* if it has the form:

$$\mathtt{r}: \mathtt{p}(..., \mathtt{X}, ...) \leftarrow \quad ..., \mathtt{next}(\mathtt{X}), ...$$

Then p is said to be a *stage predicate* and the argument position where X appears is said to be *a stage argument* for p. When p is a stage predicate, with X occupying its stage argument, and there exists a rule of the form,

$$\mathtt{r}': \mathtt{q}(..., \mathtt{X}, ...) \leftarrow \quad ..., \mathtt{p}(..., \mathtt{Y}, ...), ...$$

then, q is also called a stage predicate and X denotes its stage argument. A rule such as r' which does not have choice goals will be called a *flat* rule. The recursive application of these definitions produces all possible stage predicates, and their arguments. A maximal set of mutually recursive predicates defines a *recursive clique*. Such a clique is called a *stage clique* when the following conditions are satisfied:

- each recursive predicate in the clique is a stage predicate, with exactly one stage argument,
- any two recursive rules defining a predicate in the clique must be of the same kind (i.e., either they are both next rules or flat rules, but not a mixture of them).

An *instance* of a rule r is obtained by assigning values from the Herbrand universe to the variables of r. When this assignment satisfies all the equality and inequality goals, then we obtain an *interpreted instance* of r. We can now introduce the notion of stage-stratified recursive cliques. Let r be a rule in the stage clique (e.g., a `next` rule) and let r' denote the rule obtained from r by (1) rewriting the `next` goal, (2) eliminating the `choice` goals and (3) rewriting the `least` goals. Now, consider the stage predicates in r' and their stage arguments. Then, if for every interpreted instance of r' the value of the stage argument in the head is $\geq$ than each stage arguments in the tail, then we say that r is *stage-stratified*. When all the inequalities hold in the strict sense, we will say that the rule is *strictly stage-stratified*. For instance, if r is the next rule in Example 4, then r' is:

$$\begin{aligned} \mathtt{prm}(\mathtt{X}, \mathtt{Y}, \mathtt{C}, \mathtt{I}) \leftarrow \quad & \mathtt{prm}(\_, \_, \_, \mathtt{I1}), \mathtt{I} = \mathtt{I1} + 1 \\ & \mathtt{new\_g}(\mathtt{X}, \mathtt{Y}, \mathtt{C}, \mathtt{J}), \ \mathtt{J} < \mathtt{I}, \\ & \neg(\mathtt{prm}(\_, \_, \_, \mathtt{I1}'), \mathtt{I} = \mathtt{I1}' + 1 \\ & \mathtt{new\_g}(\mathtt{X}', \mathtt{Y}', \mathtt{C}', \mathtt{J}'), \ \mathtt{J}' < \mathtt{I}, \\ & \mathtt{C}' < \mathtt{C}). \end{aligned}$$

Here, the stage variables in the body are $I1$, $I1'$, J and J'. The values of these stage variables are all less than that of I in any interpreted occurrence of this rewritten rule. Thus we conclude that the original rule is strictly stage-stratified. (Observe the role of `least(C, I)` in ensuring this result; if we replace this goal by `least(C, _)`, the stage-stratification is lost.) Likewise, we conclude that the *flat* rule of Example 4 is stage-stratified.

Therefore we have the following definition: a clique is said to be *stage-stratified*, if it is a stage clique and

- `next` rules are strictly stage-stratified, and
- Each positive goal in flat rules is stage stratified w.r.t. the stage variable in the head and each negated goal is strictly stage stratified w.r.t. the stage variable in the head.

A program will be said to be stage-stratified if it consists of Horn clauses and stage-stratified cliques. Example 4 illustrating Prim's algorithm is an example of a stage-stratified program. Obviously, stage-stratified programs are locally stratified modulo choice, and there exist one or more stable models for each stage-stratified program. Moreover, we have the following important result[2]:

Theorem 1: *Let P be a stage-stratified program. Then, every set of facts produced by the Choice Fixpoint is a stable model for P.*

Proof [outline]: Consider any rule instance r used by the choice fixpoint procedure in constructing a candidate stable model M. M is a model for P. To prove that it is a stable model, observe, that, if $\neg g$ is a goal in r then g is not in M. (If g is a $diffChoice$ predicate, then g is not in M because of the symmetry in the rules defining $diffChoice$; if g is anything else, the conclusion follows from the stage stratification assumption.) Thus as we apply the stability transformation to r, using M, [3]), we find all its negative goals are satisfied, and are thus removed from r yielding a positive rule r'. Thus the positive program produced by the stability transformation has a minimal model that is a superset M. □

A program P containing negation, extrema, choice and next goals will be said to be a *next-Datalog* program if the only instances of function symbols (or evaluatable predicates) in the program are those used in the definition of `next`. The *reduced* Herbrand universe and base for these programs are those obtained neglecting these function symbols. Then, it follows that:

Theorem 2: *Let P be a stage-stratified next-Datalog program. Then,*

- *The Reduced Herbrand Universe and Reduced Herbrand Base of P are finite.*
- *The Choice Fixpoint procedure is (non-deterministically) complete.*

- *The data complexity of computing a stable model for P is polynomial time.*

□

In our examples, the programs satisfy the following additional property: $Q^{\infty}(\gamma(S)) = Q(\gamma(S))$. These programs will be called *alternating stage-stratified* programs. Here the computation of stable models, basically reduces to the traditional fixpoint, with the rule firing alternating between next rules (γ) and flat rules (Q).

Alternating Stage-Choice Fixpoint
begin
  $S' := \emptyset$;
  repeat
    $S := S'$;
    $S' := Q(\gamma(S))$;
  until $S' = S$
end.

The following completeness theorem holds for alternating stage-stratified programs.

Theorem 3: The Alternating Stage-Choice Fixpoint procedure is (non-deterministically) complete for alternating stage-stratified programs. □

This approach can be also extended to all programs where the flat rules alone (i.e., without the next rules) do not define a recursive clique. In this case, $Q^{\infty} = Q^n$, with n equal to or less than the number of flat rules in the clique. Then it is possible to order these rules in such a way, that Q^n can be computed in one pass, by letting the results of one flat rule feed into the next rule. In this case the computation alternates between the firing of the next rules and the firing of a chain of n flat rules.

5 Examples

In this section, we present some examples to show that many classical greedy algorithms may be expressed as stage stratified programs.

Example 5: In this example, we sort a relation. Given a set of tuples `p(X,C)` we generate a new set of tuples of the form `sp(X,C,I)` such that for any two tuples $sp(a, c, i)$ and $sp(b, c', j)$, $i \leq j$ iff $c \leq c'$. Intuitively, at each step the smallest tuple from the remaining set of tuples is selected and inserted into sp.

```
sp(nil, 0, 0).
sp(X, C, I) ← next(I), p(X, C), least(C, I).
```
□

Example 6: Huffman codes is a prefix text-compression technique, based on assigning shorter codes to the characters appearing more frequently. The following program generates a tree, called Huffman tree, that can be scanned to generate the codes. The base predicate `letter(X,C)` gives the relative frequency `C` with which the character `X` appears in the text. The predicate `h(X,C,I)` denotes that `X` is a subtree of the Huffman tree with cost `C` and is obtained when the value of the stage variable is `I`. The predicate `feasible(t(X,Y),C,I)` indicates that `X` and `Y` are both subtrees of the Huffman tree and the tree constructed by having `X` as the left child and `Y` as the right child has a cost of `C`. The functor `t` is the tree constructor.

```
h(X, C, 0) ←          letter(X, C).
h(t(X, Y), C, I) ←    next(I), feasible(t(X, Y), C, J),
                      J < I, least(C),
                      choice(X, I), choice(Y, I).

feasible(t(X, Y), C, I) ←  h(X, C1, J), h(Y, C2, K),
                           ¬subtree(X, L1), L1 < I,
                           ¬subtree(Y, L2), L2 < I,
                           I = max(J, K), X ≠ Y,
                           C = C1 + C2.

subtree(X, I) ←  h(t(X, _), _, I).
subtree(X, I) ←  h(t(_, X), _, I).
```
□

Example 7: The problem is to find a maximal matching in a directed graph with the minimum cost. A matching is a set of arcs such that no two arcs in the set share a common vertex. The cost of a matching is the sum of the costs of the arcs in the matching. The predicate `g(X,Y,C)` describes the arcs and the cost of the arcs in the given graph.

```
matching(nil, nil, 0, 0).
matching(X, Y, C, I) ←  next(I), g(X, Y, C),
                        least(C, I), choice(Y, X),
                        choice(X, Y).
```
□

Computation of Sub-Optimals: Greedy algorithms often provide efficient approximate solutions to NP-hard problems such as the *Traveling Salesperson Problem.*

Given a complete undirected graph, a greedy solution starts by selecting the arc with minimum cost (by the exit rule defining the `tsp_chain` predicate). Then, the arc with minimum cost from node X to node Y is selected, provided that an arc with starting node Y has not been previously selected. We repeat this operation until a Hamiltonian path is obtained.

```
tsp_chain(X, Y, C, 1) ←  least_arcs(X, Y, C),
                         choice((), (X, Y)).
tsp_chain(X, Y, C, I) ←  next(I), new_g(X, Y, C, J),
                         I = J + 1, least(C, I),
                         choice(Y, X).
```

```
new_g(X, Y, C, J) ← tsp_chain(_, X, _, J), g(X, Y, C).
least_arcs(X, Y, C) ← g(X, Y, C), least(C).
```
□

Other greedy algorithms that have been expressed as stage-stratified programs, include the *set covering* problem, the *convex hull* problem, several *scheduling algorithms* and others [2].

6 Complexity Analysis

In this section, we consider the efficient implementation of the Alternating Stage-Stratified fixpoint for a subclass of alternating stage stratified programs. Suppose that the body of a *next* rule r is of the form

$$\texttt{next(I)}, \texttt{p}(\bar{\texttt{X}}, \texttt{J}), [\texttt{J} < \texttt{I}, \texttt{least(C, I)}], [\texttt{choice } \textit{predicates}],$$

(where the predicates within brakets could be missing). Then, we say that two p-facts f_1 and f_2 are r-congruent, if they agree on all arguments of p except possibly the stage arguments, the cost argument (i.e., the first argument in a `least` predicate), and the functionally determined attributes of choice.

The combination of `least` and `next` in a rule r may be efficiently implemented as a data structure $D_r = (R_r, Q_r, L_r)$, where Q_r denotes the priority queue of the candidate solutions to the least predicate, L_r consists of those facts which have been used to fire the rule in the previous iteration and R_r contains those tuples which cannot be candidate solutions (or Redundant tuples). The operations supported are *insertion* and *retrieve least* from D_r.

1. The insertion operation takes a p-fact f and begins with searching for a fact $f_1 \in Q_r \cup L_r$ such that f_1 and f are r-congruent. If $f_1 \in L_r$ then f is inserted in R_r, which is maintained as a simple set. If $f_1 \in Q_r$, then f is inserted in R_r if the cost of f_1 is less than the cost of f; otherwise, f_1 is deleted from Q_r and inserted in R_r and f is inserted in Q_r. If there is no such f_1 then f is inserted into Q_r.

2. The *least* operation is supported by maintaining Q_r as a priority queue. This operation deletes a smallest element retrieved from Q_r and also inserts into L_r. This fact is then returned as the *chosen* value. In case of several least elements, exactly one is chosen. If the priority queue contains n elements, then the complexity of this operation is $O(log\ n)$.

If the next rule r does not have a least predicate, then Q_r may be maintained as a simple set and not as a priority queue. The retrieve least operation is then a retrieve *any* operation. Assuming availability of indices, the insertion operation described above is $O(log(|Q|))$.

Prim's Algorithm: Complexity of Example 4

Consider the program expressing Prim's algorithm presented at the end of Section 3. The (R, Q, L) structure is maintained for the predicate *new_g*, where the congruence is defined as $new\_g(a, b, c, i) \cong new\_g(e, f, g, j)$ if $a = e$. Let n and e be the number of vertices and edges respectively in the graph. The priority queue, Q is a set of edges and hence its size is bounded by e. The maximum number of tuples in *new_g* is bounded by e. Thus, insertion into the queue takes $O(e \times log\ e)$. Finally, assuming indices, the cost to compute one tuple of *new_g* is constant. Hence, the worst case time complexity of our implementation of Prim's algorithm is $O(e \times log\ e)$, which is comparable to the classical complexity of $O(e \times log\ n)$.

Sorting: Complexity of Example 5

The implementation of Example 4 has a complexity bound of $O(n \times log\ n)$. The predicate p is first stored as a priority queue (at a cost of $O(n \times log\ n)$ if n is the cardinality of p). At each iteration, the smallest tuple of Q is moved into L, this costs $O(n \times log\ n)$. Hence, the complexity of this algorithm is $O(n \times log\ n)$. We notice that although the program expresses an "insertion sort" like algorithm, the fixpoint algorithm implements a "heap-sort".

Matching: Complexity of Example 7

The implementation of Example 7 has a complexity of $O(e \times log\ e)$ where e is the number of arcs in the graph. The tuples of *arc* are stored by using a priority queue Q. The cost of building Q is $O(e \times log\ e)$. At each step, the least element is selected from Q and, if it satisfies the choice conditions, then it is moved into L, otherwise it is moved into R. The maximum number of tuples in Q is e and the cost of extracting one tuple from Q is $O(log\ e)$. Hence, the total cost of the algorithm is bounded by $O(e \times \log\ e)$.

7 Conclusion

The results presented in this paper are still preliminary in nature, and we are in the process of extending and refining these results and deploying them in actual systems. [2]. An intriguing possibility consists in the automatic detection of declarative programs that can be implemented by greedy algorithms. Consider the following naive version of the matching problem of Example 6. The problem of finding a minimum cost maximal matching is posed as a post-condition on *matching*, defined as follows:

```
opt_matching(C) ← a_matching(C), least(C).
a_matching(C) ← matching(X, Y, C, I), most(I).
matching(nil, nil, 0, 0).
matching(X, Y, C, I) ← next(I), new_arc(X, Y, C, J),
                       I = J + 1, choice(Y, X),
                       choice(X, Y).
```

```
new_arc(X, Y, C, J) ← new_arc(_, _, C1, J),
                      g(X, Y, C2),  C = C1 + C2.
```

This program can be transformed into the efficient solution of Example 7, in a fashion similar to propagation of extrema predicates into recursion presented in [1]. The program above corresponds to a *partition matroid*, while Kruskal's algorithm for minimum spanning tree, presented in Example 8 is a *graphic matroid*. The problem of deriving simple sufficient conditions for the propagation of least into stage stratified programs based on Matroid Theory [12] (or its generalizations such as *greedoid* [6] and *Matroid embedding* [5] is left open.

Another interesting direction of research is the automatic compilation of stage programs into efficient data structures which extend the generality of technique presented in Section 6.

Finally, research is needed to extend the class of stage stratified programs to include porgrams in which flat rules are not necessarily strictly stratified. The following example presents a minimum spanning tree algorithm due to Kruskal. Although the negation in flat rules are not strictly stratified, the stable model of this program gives a minimum spanning tree of the graph, as desired. This example also discusses how data structures leading to an efficient implementation may be designed, similar to the discussion in Section 6.

Example 8: Kruskal's algorithm computes a minimum spanning tree on a graph `g`. Initially, it associates each node `X` with a connected component `K` (predicate `comp`). As the algorithm proceeds, at each step `I`, is selected a new arc `(X,Y,C)` such that the last identifiers associated with `X` and `Y` are distinct with minimum cost. Let `(A,B,C)` be the arc selected at step `I`, and let `J` and `K` be the last identifiers used to denote the components associated, respectively, with `A` and `B`; then the elements of the component `J` are associated with the component `K` (recursive rule of `comp`). The predicate `most` is the dual of `least`. Notice that the predicates `most` and `least` are used in the same clique but no ambiguity arises since they refer to different arguments.

```
kruskal(X, Y, C, 0) ← g(X, Y, C), least(C),
                      choice((), (X, Y)).
kruskal(X, Y, C, I) ← next(I), g(X, Y, C),
                      last_comp(X, J, I1),
                      last_comp(Y, K, I1),
                      J ≠ K,  I1 < I,  least(C).

last_comp(X, J, I) ← comp(X, J, I1), I1 ≤ I,
                     most(J, X).

comp(X, K, 0) ← comp0(X, K).
comp(X, K, I) ← kruskal(A, B, C, I),
                last_comp(A, J, I1),
                last_comp(B, K, I2),
                last_comp(X, J, I1).

comp0(nil, 0).
comp0(X, K) ← next(K), node(X).
```

□

Kruskal: Complexity of Example 8

The implementation of Example 7 has a complexity bound of $O(e \times n)$ where e and n are respectively the number of edges and nodes in the graph. The computation of the predicate `comp` has cost $O(n)$. The edges of the graph `g` are stored by using a priority queue Q, at cost $O(e \times log\ e)$. At each step, the edge with minimum cost connecting two distinct components is selected from Q. If the edge with minimum cost connects two nodes of the same component, then it is redundant and is moved into R. The tuples of `comp` are also stored both in Q and R. When the value of the component of a node X is updated the old tuple is moved into R because it will never be used. The cost of updating the nodes of a component is $O(n)$. The total cost is then $O(e \times n)$. Notice that the classical procedural method takes $O(e \times log\ e)$. The difference is due to the fact that the classical algorithm 'merge' the smallest component into the 'largest'.

References

[1] S. Ganguly, S. Greco, and C. Zaniolo. Minimum and maximum predicates in logic programming. In *Proceedings of the Tenth ACM Symposium on Principles of Database Systems*, pages 154–163, 1991.

[2] S. Ganguly, S. Greco, and C. Zaniolo. Greedy algorithms in deductive databsases. Research report, in preparation, UCLA, 1992.

[3] M. Gelfond and V. Lifschitz. The stable model semantics of logic programming. In *Proceedings of the Fifth Intern. Conference on Logic Programming*, pages 1070–1080, 1988.

[4] F. Giannotti, D. Pedreschi, D. Saccà, and C. Zaniolo. Nondeterminism in deductive databases. In *Proc. 2nd Int. Conf. on Deductive and Object-Oriented Databases*, 1991.

[5] P. Helmann, B.M.E. Moret, and H.D. Shapiro. Exact caracterization of greedy structures. Research Report CS89-11, Univ. of New Mexico, 1989.

[6] B. Korte and L. Lovàsz. Greedoids - a structural framework for the greedy algorithm. In W. R. Pulleyblank, editor, *Progress in Combinatorial Optimization*, pages 221–243. Academic Press, 1984.

[7] R. Krishnamurthy and S. Naqvi. Non-deterministic choice in Datalog. In *Proceedings of the 3rd International Conference on Data and Knowledge Bases*, 1988.

[8] T. Przymusinski. On the declarative and procedural semantics of stratified deductive databases. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 193–216. Morgan-Kaufman, Los Altos, CA, 1988.

[9] D. Saccà and C. Zaniolo. Stable models and non-determinism in logic programs with negation. In *Proceedings of the Ninth ACM Symposium on Principles of Database Systems*, pages 205–217, 1990.

[10] D. Saccà and C. Zaniolo. Deterministic versus non-deterministics semantics in deductive databases. In *Submitted for Pubblication*, 1991.

[11] A. Van Gelder, K.A. Ross, and J.S. Schlipf. The well-founded semantics for general logic programs. *Journal of ACM*, 38(3):620–650, 1991.

[12] D.J.A. Welsh, editor. *Matroid Theory*. Academic Press, 1976.