Greedy Algorithms in Datalog with Choice and Negation

Sergio Greco
Dip. Elettr. Informatica Sist.
Università della Calabria
87030 Rende, Italy
greco@si.deis.unical.it

Carlo Zaniolo
Computer Science Dept.
Univ. of California at Los Angeles
Los Angeles, CA 90024
zaniolo@cs.ucla.edu

Abstract

In the design of algorithms, the greedy paradigm provides a powerful tool for solving efficiently classical computational problems, within the framework of procedural languages. However, expressing these algorithms within the declarative framework of logic-based languages has proven a difficult research challenge. In this paper, we extend the framework of Datalog-like languages to obtain simple declarative formulations for such problems, and propose effective implementation techniques to ensure computational complexities comparable to those of procedural formulations. These advances are achieved through the use of the `choice` construct, extended with preference annotations to effect the selection of alternative stable-models and nondeterministic fixpoints. We show that, with suitable storage structures, the differential fixpoint computation of our programs matches the complexity of procedural algorithms in classical search and optimization problems.

1 Introduction

The problem of finding efficient implementations for declarative logic-based languages represents one of the most arduous and lasting research challenges in computer science. The interesting theoretical challenges posed by this problem are made more urgent by the fact that extrema and other non-monotonic constructs are needed to express many real-life applications, ranging from the 'Bill of Materials' to graph-computation algorithms.

Significant progress in this area has been achieved on the semantic front, where the introduction of the well-founded model semantics and stable-model semantics allows us to assign a formal meaning to most, if not all, programs of practical interest. Unfortunately, the computational problems remain largely unsolved: various approaches have been proposed to more effective computations of well-founded models and stable models [20, 7], but these fall well short of matching the efficiency of classical procedural solutions, such as, say, algorithms that find shortest paths in graphs. In general, it is known that determining whether a program has a stable model is NP-complete [14].

Therefore, in this paper we propose a different approach: while, at the semantic level, we strictly adhere to the formal declarative semantics of logic programs with negation, we also allow the use of extended non-monotonic constructs with first order semantics to facilitate the task of programmers and compilers alike. This entails simple declarative formulations and nearly optimal executions for large classes of problems, that are normally solved using greedy algorithms.

Greedy algorithms [15] are those that solve a class of optimization problems, using a control structure of a single loop, where, at each iteration some element judged the 'best' at that stage is chosen and it is added to the solution. The simple loop hints that these problems are amenable to a fixpoint computation. The choice at each step calls attention to mechanisms by which nondeterministic choices can be expressed in logic programs. This framework also provides an opportunity of making, rather than blind choices, choices based on some heuristic criterion, such as greedly choosing the least (or most) among the values at hand when seeking the global minimization (or maximization) of the sum of such values. Following these hints, this paper introduces primitives for choice, and greedy selection, and show that classical greedy algorithms can be expressed using them.

A significant amount of excellent previous work has investigated the issue of how to express in logic and compute efficiently greedy algorithms, and, more in general, classical algorithms that require non-monotonic constructs. An incomplete list include work by [19, 5, 18, 21, 6]. In this paper we introduce a treatment for greedy algorithms that is significant simpler and more robust than previous approaches (including that of Greco, Zaniolo and Ganguly [11]); it also treats all aspects of these algorithms, beginning from their intuitive formulation, and ending with their optimized expression and execution.

For space limitation the proofs of our results are omitted and they can be found in the full version of the paper [12].

2 Nondeterministic Reasoning

Say that, with relation `student(Name, Major, Year)`, our university database contains the relation `professor(Name, Major)`. In fact, say that our toy database contains only the following facts:

```
student('JimBlack', ee, senior).        professor(ohm, ee).
                                        professor(bell, ee).
```

Now, the rule is that the major of a student must match his/her advisor's major area of specialization. Then eligible advisors can be computed as follows:

$$\texttt{elig\_adv(S,P)} \leftarrow \ \texttt{student(S,Majr,Year)},\ \texttt{professor(P,Majr)}.$$

This yields

$$\texttt{elig\_adv('JimBlack',ohm)}.$$
$$\texttt{elig\_adv('JimBlack',bell)}.$$

But, since a student can only have one advisor, the goal $\texttt{choice((S),(P))}$ must be added to force the selection of a unique advisor, out of the eligible advisors, for a student.

Example 1. *Computation of unique advisors by choice rules*

$$\begin{aligned}\texttt{actual\_adv(S,P)} \leftarrow\ & \texttt{student(S,Majr,Yr)},\ \texttt{professor(P,Majr)},\\ & \texttt{choice((S),(P))}.\end{aligned}$$

The computation of this rule gives for each student S a unique professor P □

The goal $\texttt{choice((S),(P))}$ can also be viewed as enforcing a functional dependency (FD) $\texttt{S} \rightarrow \texttt{P}$; thus, in $\texttt{actual\_adv}$, the second column (professor name) is functionally dependent on the first one (student name).

The result of executing this rule is *nondeterministic*. It can either give a singleton relation containing the tuple $(\texttt{'JimBlack'},\texttt{ohm})$ or that containing the tuple $(\texttt{'JimBlack'},\texttt{bell})$.

A program where the rules contain choice goals is called a *choice program*. The semantics of a choice program P can be defined by transforming P into a program with negation, $foe(P)$, called the *first order equivalent* of a choice program P. $foe(P)$ exhibits a multiplicity of stable models, each obeying the FDs defined by the choice goals. Each stable model for $foe(P)$ corresponds to an alternative set of answers for P and is called a *choice model* for P. $foe(P)$ is defined as follows:

Definition 1. The *stable version* $foe(P)$ of a choice program P is obtained by the following transformation. Consider a choice rule r in P:

$$r : A \leftarrow B(Z),\ choice((X_1),(Y_1)),\ \ldots,\ choice((X_k),(Y_k)).$$

where,

(i) $B(Z)$ denotes the conjunction of all the goals of r that are not choice goals, and

(ii) X_i, Y_i, Z, $1 \leq i \leq k$, denote vectors of variables occurring in the body of r such that $X_i \cap Y_i = \emptyset$ and $X_i, Y_i \subseteq Z$.

Then the original program P is transformed as follows:

1. Replace r with a rule r' obtained by substituting the choice goals with the atom $chosen_r(W)$:

   $$r' : A \leftarrow B(Z),\ chosen_r(W).$$

   where $W \subseteq Z$ is the list of all variables appearing in choice goals, i.e., $W = \bigcup_{1 \leq j \leq k} X_j \cup Y_j$.

2. Add the new rule

   $$chosen_r(W) \leftarrow B(Z),\ \neg diffChoice_r(W).$$

3. For each choice atom $choice((X_i),(Y_i))$ $(1 \leq i \leq k)$, add the new rule

   $$diffChoice_r(W) \leftarrow chosen_r(W'),\ Y_i \neq Y_i'.$$

   where (i) the list of variables W' is derived from W by replacing each $A \notin X_i$ with a new variable A' (i.e., by priming those variables), and (ii) $Y_i \neq Y_i'$ is true if $A \neq A'$, for some variable $A \in Y_i$ and its primed counterpart $A' \in Y_i'$.

The stable version of Example 1 is given in Example 2, which can be read as a statement that a professor will be assigned to each student if a different professor has not been assigned to the same student.

Example 2. The stable version of the rule in Example 1

```
actual_adv(S,P) ←     student(S,Majr,Yr), professor(P,Majr),
                      chosen(S,P).
chosen(S,P) ←         student(S,Majr,Yr), professor(P,Majr),
                      ¬diffChoice(S,P).
diffChoice(S,P) ←     chosen(S,P′), P ≠ P′.
```

In general, the program $foe(P)$ generated by the transformation discussed above has the following properties[8]:

- $foe(P)$ has one or more total stable models.
- The `chosen` atoms in each stable model of $foe(P)$ obey the FDs defined by the choice goals.

The stable models of $foe(P)$ are called *choice models* for P.

While the topic of operational semantics for choice Datalog programs will be further discussed in Section 4, it is clear that they can be implemented efficiently. Basically, the `chosen` atoms need to be produced one-at-a-time and memorized in a table. The `diffchoice` atoms need not be computed and stored; rather, the goal `¬diffchoice` can simply be checked dynamically against the table `chosen`. Since these are simple operations (actually quasi constant-time if an hash table is used), it follows that choice Datalog programs can be computed in polynomial time, and that rules with choice can be evaluated as efficiently as those without choice.

3 Computing with Choice

The introduction of choice in Datalog has a significant impact on the expressive power of the language, since it allows us to express *nondeterministic* queries that cannot be expressed in Datalog with stratified negation, and *deterministic* queries that could not be expressed otherwise [10, 8].

In fact, the expressive power of the choice construct has been studied in [9, 10], where it is shown that it is more powerful than other nondeterministic constructs, including the witness operator [2], and the original version of choice proposed in [13], which is called static-choice, to distinguish from the dynamic choice used here [8]). For instance, it has been shown in [8], that the task of ordering a domain or computing whether it contains an even number of elements (parity query) cannot performed by positive programs with static choice or the witness operator [2].

In the rest of the paper, we will concentrate on nondeterministic queries. In particular, our previous advisor example, can be modified using more sophisticated criteria are used to match students with candidate advisors. Say that the application of these criteria has yielded a relation `candidate(Student, Advisor, D)`, which for each new `Student` gives several professors suitable to become their advisors (with and additional column `D` denoting dates at which the new advisory engagement can be initiated—thus there might be several values of `D` for each student-professor pair.) Then our matching problem can be expressed as follows:

Example 3. *Matching in a directed graph*

```
advisor(St, Adv) ←  candidate(St, Adv, D),
                    choice((St), (Adv)), choice((St), (D)).
```
□

By viewing candidate as a graph from `St` to `Adv` we immediately see that we have here an instance of a bi-partite graph-matching problem. This also defines the optimization problem discussed in Section 5, by requiring that the sum of all `D`s be minimized or maximized[1].

Example 4. *Rooted spanning tree.* We are given an undirect graph where an edge joining two nodes, say x and y, is represented by means of two fact $g(x, y, c)$ and $g(y, x, c)$, where c is the cost. A spanning tree in the graph, starting from the source node `a`, can be expressed by means of the following program:

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

[1] Given that the pair `St` → `Adv`, `St` → `D` is equivalent to `St` → `Adv,D`, the last rule could also have been written as:

```
advisor(St,Adv) ← candidate(St,Adv,D),  choice((St),(Adv,D)).
```

To illustrate the presence of multiple total choice models for this program, take a simple graph consisting of the following arcs:

```
g(a,b,1).   g(b,a,1).
g(b,c,2).   g(c,b,2).
g(a,c,3).   g(c,a,3).
```

Several outcomes are possible for this program. For instance, after the exit rule adds st(root, a, 0), the recursive rule could add g(a, b, 1) and st(a, c, 3) along with the two tuples chosen(b, a, 1) and chosen(c, a, 3) in the chosen table. No further arc can be added after those, since the addition of st(b, c, 2) or st(c, b, 2) would violate the FD that follows from choice((Y), (X)) enforced through the chosen table. However, since st(root, a, 0), was produced by the first rule (the exit rule), rather than the second rule (the recursive choice rule), the table chosen contains no tuple (root, a). Therefore, to avoid the addition of st(c, a, 3) or st(b, a, 1), the goal $Y \neq a$ was added to the recursive rule.

By examining all possible solutions, we conclude that this program has three different choice models, for which we list only the st-atoms, below:

1. st(a, b, 1).   st(b, c, 2).
2. st(a, b, 1).   st(a, c, 3).
3. st(a, c, 3).   st(c, b, 2).   □

Example 5. *Single-Source Reachability.* Given a direct graph where the arcs are stored by means of tuples of the form $g(x, y, c)$, the set of nodes reachable from a node a can be defined by the following program:

```
reach(a,0).
reach(Y,C) ←  reach(X,C1), g(X,Y,C2), Y ≠ a,
              C = C1 + C2, choice((Y),(C)).
```
□

Once the cost arguments are eliminated from these rules, we obtain the usual transitive-closure-like program, for which the fixpoint computation terminates once all nodes reachable from node a are found, even if the graph contains cycles. However, if the choice goal were eliminated, the program of Example 5 could become nonterminating on a cyclic graph. This illustrates a first useful application of the choice constraint.

In the next example, we have a complete undirected labeled graph G, represented by facts g(x, y, c), where the label c typically represents the cost of the edge. A simple path of G is a path passing through each node at most once. A Hamiltonian path is a simple path reaching each node in the graph. Then, a simple path can be constructed as follows:

Example 6. *The Hamiltonian path problem.*

```
s-path(root,X,0) ←  g(X,_,_), choice((),X).
s-path(X,Y,C) ← s-path(_,X,_), g(X,Y,C), s-path(root,Z,_), Y ≠ Z,
                choice((X),(Y)), choice((Y),(X)), choice((Y),(C)).
```
□

When G is a complete graph, the simple path produced by this program is Hamiltonian (i.e., touches all the nodes). In many applications, we need to find a minimum-cost Hamiltonian path; this is the Traveling Salesman Problem (TSP) discussed in Section 5.

4 Fixpoint Semantics for Choice Programs

Let I be an interpretation for a program P; the *immediate consequence operator* $T_P(I)$ is defined as the set containing the heads of each rule $r \in ground(P)$ s.t. all positive goals of r are in I, and none of the negated goals of r, is in I. For a choice program P, with first order equivalent $foe(P)$, let us denote by T_{P_C} the immediate consequence operator associated with the rules defining the predicate `chosen` in $foe(P)$ (these are the rules with the $\neg$`diffchoice` goals) and let T_{P_D} denote the immediate consequence for all the other rules in $foe(P)$ (for positive choice programs these are Horn clauses).

Therefore, we have that, for any interpretation I of $foe(P)$:

$$T_{foe(P)}(I) = T_{P_D}(I) \ \cup \ T_{P_C}(I).$$

Following [9] we can now introduce a general operator for computing the nondeterministic fixpoints of a choice program P. The stable models of $foe(P)$ are called *choice models* for P, and the choice goals define the functional dependencies FD_P:

Definition 2. Given a choice program P, its *nondeterministic immediate consequence operator* Ψ_P is a map from an interpretation of $foe(P)$ to a set of interpretations of $foe(P)$ defined as follows:

$$\Psi_P(I) = \{ \ T_{P_D}^{\uparrow\omega}(J) \mid \ J \in \Gamma_P(I) \ \} \tag{1}$$

where:

$$\begin{aligned} \Gamma_P(I) \quad = \quad & \{ I \cup \Delta I \ \mid \ \emptyset \subset \Delta I \subseteq T_{P_C}(I) \setminus I \text{ and } I \cup \Delta I \models FD_P \ \} \\ & \cup \{ \ I \ \mid \ T_{P_C}(I) \setminus I = \emptyset \ \}. \end{aligned} \tag{2}$$

Where $I \cup \Delta I \models FD_P$ denotes that $I \cup \Delta I$ satisfies the dependencies in FD_P. $\Box$

Therefore, the Ψ_P operator, is basically the composition of two operators. Given an interpretation I, the first operator computes all the admissible subsets of $\Delta I \subseteq T_{P_C}(I)$, i.e., those where $I \cup \Delta I$ obeys the given FDs; the second operator derives the logical consequence for each admissible subset using the ω-power of T_{P_D}.

The definition of $\Gamma_P(I)$ is such that ΔI is not empty iff $T_{P_C}(I) \setminus I$ is not empty; thus, if there are possible new choices, then at least one has to be taken.

The Ψ_P operator formalizes a single step of an ideal bottom-up computation of a choice program. Instead of defining the powers of Ψ_P, it is technically more convenient to define directly the notion of a nondeterministic computation based on the Ψ_P operator.

Definition 3. *Given a choice program P, an* inflationary choice fixpoint computation *for P, is a sequence $\langle I_n \rangle_{n \geq 0}$ of interpretations such that:*

i. $I_0 = \emptyset$,

ii. $I_{n+1} \in \Psi_P(I_n)$, *for* $n \geq 0$. □

Inasmuch as every sequence $\langle I_n \rangle_{n \geq 0}$ is monotonic, it has a unique limit for $n \to \infty$; this limit will be called an *inflationary choice fixpoint* for the choice program P. Thus, we have the following result.

Theorem 1. *[8] Let P be a Datalog program with choice, and M a Herbrand interpretation for $foe(P)$. Then M is a choice model for P iff M is an inflationary choice fixpoint for P.* □

Moreover, the inflationary choice fixpoint is *sound* (every result is a choice model) and *complete* (for each choice model there is some inflationary choice fixpoint computation producing it). For logic programs with infinite Herbrand universe, an additional assumption of *fairness* is needed to ensure completeness [9]. As customary for database queries, computational complexity is evaluated with respect to the size of the database.

Theorem 2. *[8] Let P be a choice Datalog program. Then, the data complexity of computing a choice model for P is polynomial time.* □

Therefore, for a choice Datalog program, P, the computation of one of the stable models for $foe(P)$ can be performed in polynomial time using the Choice Fixpoint Computation. This contrast with the general untractability of finding stable models for general programs: in fact, we know that checking if a Datalog program with negation has a stable model is NP-complete [14].

Therefore the choice construct allows us to capture a special subclass of programs that have a stable model semantics but are amenable to efficient implementation and are appealing to intuition. Implementing these programs only requires memorization of the *chosen* predicates; from these, the *diffchoice* predicates can be generated on-the-fly, thus eliminating the need to store `diffchoice` explicitly. Moreover, the model of memorizing tables to enforce functional dependencies provides a simple enough metaphor for a programmer to make effective usage of this construct without having to become cognizant on the subtleties of non-monotonic semantics.

4.1 Greedy Choice

Definition 2 leaves quite a bit of latitude in the computation of Δ (Equation 2). This freedom can be used to select Δs that have additional properties. In particular, we want to explore specializations of this concept that trade nondeterministic completeness (which is only of abstract interest to a programmer) in return for very concrete benefits, such as expressive power and performance. For instance, in the specialization called *Eager Choice* [8], a maximal ΔI is used in Equation 2. This results in a significant increase in expressive power, as demonstrated by the fact that negation can be emulated by eager choice [8, 9].

In this paper, we introduce a new kind of specialization called *greedy choice*; this is motivated by the observation that it is frequently desirable to select a value that is the *least* (or the *most*) among the possible values and still satisfy the FDs defined by the choice atoms [11]. Therefore, with X denoting one or more variables of r, and C denoting one single variable ranging over an ordered domain, a goal `choice-least((X),(C))` in a rule r can be used to denote that the FD `choice((X),(C))` is to be satisfied using least values of `C`. Then we will write `choice-least`$((\mathtt{X}),(\mathtt{C}))$ to denote this variation of choice (`choice-most`$((\mathtt{X}),(\mathtt{C}))$ is the dual construct where the maximum available value of `C` is used at each step).

We can now define a choice-least rule as one that contains one choice-least goal (possibly with other choice goals). A program that contains one choice-least rule, and other rules with zero or more choice goals is called a *choice-least* program. The correct computation for `choice-least` program can be thus defined by specializing the nondeterministic immediate consequence operator of Definition 2 as follows.

Definition 4. *Least Immediate-Consequence Operator.*

Let P be a choice program and I an interpretation of P. Then $\Psi_{least(P)}(I)$ for P is defined as follows:

$$\begin{array}{lcl} \Theta_I & = & \{\delta \in T_{P_C}(I) \setminus I \mid I \cup \{\delta\} \models FD_P\} \\ \Gamma_{least(P)}(I) & = & \{I \cup \{\delta\} \mid \delta \in least_C(\Theta_I)\} \cup \{I \mid \Theta_I = \emptyset\} \\ \Psi_{least(P)}(I) & = & \{\, T_{P_D}^{\uparrow\omega}(J) \mid J \in \Gamma_{least(P)}(I)\,\} \end{array}$$

where $least_C(\Theta_I)$ denotes the subset of tuples of Θ_I which are minimal in the value of C. Ψ_{least} will be called the *Least Immediate-Consequence Operator.* □

Observe that Definition 4 is obtained from Definition 2 by letting $\Delta = \{\delta\}$ be a singleton set, and selecting out of all possible δs one that has the *least* value for the cost attribute C. Likewise, we have the dual definition of the *Most Immediate-Consequence Operator.*

Definition 5. Given a program P with choice and choice-least goals. An *inflationary least choice fixpoint computation* (LFC) for P, is a sequence $\langle I_n\rangle_{n\geq 0}$ of interpretations such that:

i. $I_0 = \emptyset$,

ii. $I_{n+1} \in \Psi_{least(P)}(I_n)$, for $n \geq 0$. □

Thus, all the tuples that do not violate the given FDs (including the FDs implied by least) are considered, and one is chosen that has the least value for the cost argument.

Theorem 3. *Let P be a Datalog program with choice and choice_least. Then,*

1. *every inflationary least choice fixpoint for P is a choice model for P.*
2. *every inflationary least choice fixpoint of P can be computed in polynomial time.* □

Therefore, the inflationary choice fixpoint computation restricted to making greedy selections is sound (i.e., produces only choice models) but no longer complete.

5 Greedy Algorithms

In a system that adopts a concrete semantics based on *least choice fixpoint*, a programmer will specify a `choice-least((X),(Y))` goal to ensure that only particular choice models rather than arbitrary ones are produced, through the greedy selection of the least values of `Y` at each step. Thus an optimal matching in a directed graph problem can be expressed as follows:

Example 7. *Optimal Matching in a directed graph*

```
best-advisor(St, Adv, D) ←  candidate(St, Adv, D),
                            choice((St), (Adv)),
                            choice-least((St), (D)).
```
□

Observe that this program is basically that of Example 1 after that the choice goal has been specialized to a choice-least goal.

The specialization of choice goals into `choice-least` or `choice-most`) goals yields a convenient and efficient formulation of many greedy algorithms, such as Dijkstra's shortest path and Prim's minimum-spanning tree algorithms are discussed next. (We assume that we have a undirected graphs where edges are labeled by their costs. Then, an edge between nodes `a` and `b` with cost 34 will be represented by the pair `g(a, b, 34)` and `g(b, a, 34)`.)

The algorithm for finding the minimum spanning tree in a weighted graph, starting from a source node `a`, can be simply derived from the program of Example 4 by replacing the goal `choice((Y), (C))` with `choice-least((Y), (C))` yielding the well-known Prim's algorithm.

Example 8. *Prim's Algorithm.*

$$
\begin{array}{ll}
\texttt{st(root,a,0).} & \\
\texttt{st(X,Y,C)} \leftarrow & \texttt{st(\_,X,\_),}\ \ \texttt{g(X,Y,C),}\ \ \texttt{Y} \neq \texttt{a,} \\
& \texttt{choice((Y),(X)),}\ \ \texttt{choice-least((Y),(C)).}
\end{array}
$$

□

Example 9. *Dijkstra's algorithm.*

$$
\begin{array}{ll}
\texttt{dj(a,0).} & \\
\texttt{dj(Y,C)} \leftarrow & \texttt{dj(X,}\texttt{C}_1\texttt{),}\ \ \texttt{g(X,Y,}\texttt{C}_2\texttt{),}\ \ \texttt{Y} \neq \texttt{a,} \\
& \texttt{C} = \texttt{C}_1 + \texttt{C}_2,\ \ \texttt{choice-least((Y),(C)).}
\end{array}
$$

□

Greedy algorithms often provide efficient approximate solutions to NP-complete problems; the following algorithm yields heuristically effective approximations of optimal solutions for the traveling salesperson problem [16].

Example 10. *Greedy TSP.*

Given a complete undirected graph[2], the exit rule simply selects an arbitrary node X, from which to start the search. Then, the recursive rule the greedily chooses an arc of least cost having X as its end node. In general, when the least cost arc from X to Y is selected, we must make sure that an arc with starting node Y had not been selected. The choice constraints, and the goal $\texttt{Y} \neq \texttt{Z}$ to avoid returns to the initial node, ensure that a Hamiltonian path is obtained.

$$
\begin{array}{ll}
\texttt{s-path(root,X,0)} \leftarrow & \texttt{node(X),}\ \ \texttt{choice((),X).} \\
\texttt{s-path(X,Y,C)} \leftarrow & \texttt{s-path(\_,X,\_),}\ \ \texttt{g(X,Y,C),} \\
& \texttt{s-path(root,Z,\_),}\ \ \texttt{Y} \neq \texttt{Z,} \\
& \texttt{choice((X),(Y)),}\ \ \texttt{choice((Y),(X)),} \\
& \texttt{choice-least((Y),(C)).}
\end{array}
$$

□

Observe that the program of Example 10 was obtained from that of Example 6 by replacing a choice goal with its choice-least counterpart. Therefore, a most encouraging conclusion emerges from the comparison of the algorithms of these section vis a vis those in Section 3. In fact, in Section 3, we formulated several graph problems using a transitive closure computation restricted by choice-enforced FD constraints, to ensure that the resulting graph has some topological properties (e.g., a tree, a path, a Hamiltonian cycle, etc.). In this section, we have shown that the specialization of the choice goal into a choice-least goal for the cost argument, turns the algorithms of Section 3, into the classical greedy algorithms of graph optimization problems. The TSP problem of Example 10 elucidates the role that the choice-least construct has in this regard. The solutions of many search problems rely on an heuristic guiding-function to prune the search and find an optimal or quasi-optimal solution quickly.

---

[2] Here every graph can be reduced to completed graph by simply adding edges of very high cost.

Here, we have seen that *choice-least* provides a very effective optimization heuristics for selecting between alternative choice models during the fixpoint computation. In fact, for the TSP problem choice-least is only guaranteed to deliver a polynomial-time approximation of the optimal solution requiring exponential time. However, for simpler problems, such as minimum spanning tree or single source shortest-path, the greedy heuristics is known to be optimal in most situations of practical interest (e.g., edges with positive costs). In conclusion, we have obtain a a framework for deriving and expressing greedy algorithms (such as Prim's algorithm) characterized by conceptual simplicity, logic-based semantics, and short and efficient programs.

6 Implementation and Complexity

A most interesting aspect of the programs discussed in this paper is that their stable models can be be computed very efficiently. Indeed, the exponential intractability of stable models is no longer an issue: our greedy fixpoint computations are always polynomial-time in the size of the database. Moreover, we obtain the same asymptotic complexity obtainable by expressing the algorithms in procedural languages, when comparable data structures are used.

In general computation computation consists of two phases: (i) compilation (ii) the execution phase. All compilation algorithms discussed here execute with time complexity that is polynomial in the size of the programs. Moreover, we will assume, as it is customarily done [22], that the size of the database dominates that of the program. Thus, execution costs dominate the compilation costs, which will then be disregarded in the derivation of the worst case complexities. We will use compilation techniques, such as the differential fixpoint computation, that are of common usage in deductive database systems [22]. Also we will employ suitable storage structures, such as hash tables to support search on keys, and priority queues to support choice-least and choice-most goals.

6.1 Implementation of Programs with Choice

Let us begin with the implementation of choice goals. As described in Section 4, all the arguments (variables or expressions) appearing in choice goals are stored in `chosen` tables. When a condition $\neg$`diffchoice` needs to be checked, it is checked directly against the `chosen` tables. No `diffchoice` atom is ever generated or stored.

Consider now, a choice rule from our recursive examples: for instance, take the following rule from Example 8.

$$\begin{aligned}\mathtt{st(X,Y,C)} \leftarrow\ & \mathtt{st(\_,X,\_),\ g(X,Y,C),\ Y \neq a,}\\ & \mathtt{choice((Y),(X)),\ choice((Y),(C)).}\end{aligned}$$

Since this is the only choice rule, and the set of arguments in the choice goals coincide with the head arguments, then, there is no need for a separate `chosen` rule and table; the `st` table which is used in the computation of the fixpoint can also play here the role of `chosen`. Thus, the `st` table that is used to memorize the `st`-tuples previously generated and discard duplicates in the fixpoint computation, can now be used to enforce the FD choice constraints as well. In our example, that means that `Y` (the second column of `st`) is a unique key, where no duplicate is allowed. The same observation holds for all the previous examples. The data structures needed to support search and insertion on keys are well-known. For main memory we can use hash tables, where searching for a key value, and inserting or deleting an entry can be considered constant-time operations.

Given this scenario, we can now determine the asymptotic complexity of the algorithms discussed in Section 3, under the assumption that the standard differential fixpoint techniques are used in the computation of recursive rules.

Complexity of rooted spanning-tree algorithm: Example 4

For each node `X` of the goal $\texttt{st}(\_, \texttt{X},_)$ this recursive rule finds the matching edges $\texttt{g}(\texttt{X}, \texttt{Y}, \texttt{C})$, before the choice conditions are checked. We can assume that the growing relation `st` is only traversed once, since the new `st` tuples generated during the differential fixpoint computation can being appended to the end of the relation (as in fact it is done by most deductive systems). Now, given the FD constraints, the second column in `st` becomes a key for `st`; thus, all the nodes of the graph are visited exactly once, at a cost $O(n)$. Moreover, for each `X`, all matching edges $\texttt{g}(\texttt{X}, \texttt{Y}, \texttt{C})$ must be found and inserted in `st` while checking for key violations. For properly constructed hash tables, these operations have cost linear in the number of edges of `g` visited, where an edge is visited at most once during the computation. Thus the overall cost is $O(e)$, where e is the number of tuples in `g` (i.e., the edges in the graph).

Complexity of single-source reachability algorithm: Example 5

This case is very similar to the previous case. The size of `reach` is n, or less, and its nodes are visited sequentially at a cost $O(n)$. However, in the process of generating `reach`, all the edges reachable from the source node `a` are explored by the algorithm. Thus the worst case complexity is $O(e)$.

Complexity of Hamiltonian circuit: Example 6

Again, all the edges in the graph will be visited in the worst case, yielding complexity $O(e)$, where $e = n^2$, according to our assumption that the graph is complete.

Complexity of graph matching: Example 3

Consider now the problem of matching students and advisors. This is a simpler situation since it is expressed without recursive rules. A compiler that only relies on the memorization of previous choices will end-up visiting each professor for each student, and then rejects every choice but the first one (with cost proportional to the product of the number of students and professor). Intelligent backtracking techniques have bee proposed to stop the system for looking for additional professors after a student has been matched with a professor.

6.2 Implementation of Choice-least and Choice-most Programs

These programs can be implemented efficiently via a simple extension of the well-known differential fixpoint technique used in deductive databases. Consider for instance Prim's algorithm in Example 8, with recursive rule:

$$\begin{aligned}\mathtt{st(X,Y,C)} \leftarrow\ & \mathtt{st(\_,X,\_),\ \ g(X,Y,C),\ \ Y \neq a,}\\ & \mathtt{choice((Y),(X)), choice\text{-}least((Y),(C)).}\end{aligned}$$

And recall the definition the least choice operator:

$$\begin{aligned}\Theta_I &= \{x \in T_{P_C}(I) \setminus I \ \mid\ I \cup \{x\} \models FD_P\}\\ \Gamma_P(I) &= \{I \cup \{\delta\} \ \mid\ \delta \in least_C(\Theta_I)\} \ \cup\ \{I \ \mid\ \Theta_I = \emptyset\}.\end{aligned}$$

Now, the Θ_I mapping, used in the choice fixpoint iteration, can be implemented by the following following *star rule*:

$$\mathtt{st^*(X,Y,C)} \leftarrow\ \mathtt{st(\_,X,\_),\ \ g(X,Y,C),\ \ Y \neq a, \neg st(X',Y,C').}$$

where the FD enforcement performed by the choice and choice least goals in the recursive rule, is now performed in this non-recursive rule by the negated goal. Then Γ_P finds the tuple with the least value of `C` generated in the last step. This computation can be improved in two simple ways:

1. Using differential techniques to compute the new value of `st*` from the old value of the same
2. Combining the computation of Θ_I with that of $least_C(\Theta_I)$, by discarding tuples that have the same `Y`-value as another tuple, but the larger value for `C`.

With the symbolic differentiation techniques, such as those used in the seminaive fixpoint computation, we partition `st*` between old tuples `old_st*` and the new tuples δ`st*`. Likewise, δ`st` denotes `st`-tuple added in the last step. Then, by symbolic differentiation, we obtain:

$$\begin{aligned}\mathtt{new\_st^*(X,Y,C)} \leftarrow\ & \mathtt{old\_st^*(X,Y,C), \neg \delta st(X',Y,C').}\\ \mathtt{new\_st^*(X,Y,C)} \leftarrow\ & \mathtt{\delta st(\_,X,\_),\ \ g(X,Y,C),\ \ Y \neq a, \neg st(X',Y,C').}\end{aligned}$$

Thus, the computation of $\mathtt{t^*(X,Y,C)}$ is now performed by (i) taking the old value of $\mathtt{st^*}$ and deleting tuples where the Y-values conflict with the newly selected δ`st`, and adding the new tuples implied by the addition of δst. As a result of this improvement, the arcs of our graph are not visited more than once. Thus, in a graph with e edges the overall cost of computing `st`* is $O(e)$.

Observe that that if `st`$*$ contains two tuples with the same Y-value but different C-value, only the tuple with the smaller value will be selected in Γ, and the other will be ignored. Thus, when computing Θ_I we only need to keep those tuples that have a minimal value of `C` for the same value of Y. Thus, when a new tuple $t_2 = (\mathtt{X}_2, \mathtt{Y}_2, \mathtt{C}_2)$ is generated, the current content of this table is searched for any conflicting tuple $t_1 = (\mathtt{X}_1, \mathtt{Y}_1, \mathtt{C}_1)$, with $\mathtt{Y}_1 = \mathtt{Y}_2$ If no such t_1 is found, then t_2 is stored in `st`*. But, if such a tuple is found and $\mathtt{C}_2 < \mathtt{C}_1$ then t_1 is obliterated and t_2 is instead stored in its place; but if $\mathtt{C}_2 < \mathtt{C}_1$ then t_1 is left untouched and t_2 is discarded. Furthermore, we can also brake ties, i.e., $\mathtt{C}_1 = \mathtt{C}_2$, in an arbitrary non-deterministic way. It thus follow that no two tuples in `st`$*$ will ever exist with identical Y-columns. Therefore, Y is a key for `st`$*$, which has maximum cardinality n (rather than e).

All is left to perform, therefore, is finding the least value of `C` in the relation `st`* incrementally computed as described above. When no priority queue is used the finding a least-cost tuple in `st`* is a $O(n)$ operation, since `st`* contains at most n tuples, where n is the number of nodes in the graph. Since this step is repeated n times the the overall cost of Prim's algorithm without priority queues is $O(n^2)$. If a priority queue is used to index the tuples in `st`*, then, the cost of finding the least value at each step is constant; thus it is $O(n)$ over n steps. However, including the cost of updating the priority queue, the maintenance of `st`* becomes the dominant cost. Basically, e entries might be added and deleted from the queue, which has maximum length n. Thus overall cost of $O(e \times log\ n)$. We can now apply a similar reasoning to determine the complexity of the Algorithms in Section 5.

Complexity of Prim's Algorithm: Example 8

As previously discussed the cost of the algorithm is $O(e \times log\ n)$ with a priority queue, and $O(n^2)$ without it.

Complexity of Dijkstra Algorithm: Example 9

Here too, while e additions might be made to `dj`$*$, its size never exceeds n. Thus, without a priority queue, the complexity is $O(n^2)$. With a priority queue, the worst case occurs when the queue must be update for each new edge visited. Then, the complexity is $O(e \times log\ e)$

Greedy TSP: Example 10

Assuming that we are dealing with a complete graph, then all edges are visited, and the cost is $O(n^2)$. Priority queues cannot improve this complexity.

Optimal Matching in a directed graph: Example 7

For each source node the least-cost sink node must be found. The overall cost is $O(n^2)$ without a priority queue and $O(n\ lg\ n)$ with one.

References

[1] Abiteboul S., R. Hull, V. Vianu. *Foundations of Databases.* Addison-Wesley. 1994.

[2] Abiteboul S. and V. Vianu. Datalog Extensions for Databases Queries and Updates. In *Journal of Computer and System Science*, 43, pages 62–124, 1991.

[3] Aho A.V., J.E. Hopcropt J.E., and J.D. Ullman, *The Design and analysis of Computer Algorithms.* Addison-Wesley, 1974.

[4] Apt K.R., H.A. Blair, and A. Walker. Towards a theory of declarative knowledge. In (Minker ed.) Foundations of Deductive Databases and Logic Programming, Morgan Kaufmann, 1988.

[5] S. W. Dietrich. Shortest Path by Approximation in Logic Programs. *ACM Letters on Programming Lang. and Sys.* Vol 1, No, 2, pages 119–137, June 1992.

[6] Ganguly S., S. Greco, and C. Zaniolo. Extrema Predicates in Deductive Databases. *Journal of Computer and System Science*, 1995.

[7] Gelfond M. and V. Lifschitz. The stable model semantics of logic programming. In *Proc. Fifth Intern. Conf. on Logic Programming*, 1988.

[8] Giannotti F., D. Pedreschi, D. Saccà, and C. Zaniolo. Nondeterminism in deductive databases. In *Proc. 2nd DOOD Conference*, 1991.

[9] Giannotti F., D. Pedreschi, C. Zaniolo, Semantics and Expressive Power of Non-Deterministic Constructs in Deductive, Tecnical Report C96-04, Cnuce-CNR, Pisa, 1996, (submitted for publication).

[10] Greco S., D. Saccà, and C. Zaniolo, DATALOG Queries with Stratified Negation and Choice: from P to D^P In *Proc. Fifth ICDT Conference*, 81-96, 1995.

[11] Greco S., C. Zaniolo, and S. Ganguly. Greedy by Choice. In *Proc. of the 11th ACM Symp. on Principles of Database Systems*, 1992.

[12] Greco S., and C. Zaniolo, Greedy Fixpoint Algorithms for Logic Programs with Negation and Extrema. Technical Report, 1997.

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

[14] Marek W., M. Truszczynski. Autoepistemic Logic. *Journal of ACM*, 38(3):588–619, 1991.

[15] Moret B.M.E., and H.D. Shapiro. *Algorithms from P to NP.* Benjamin Cummings, 1993.

[16] Papadimitriou C., K. Steiglitz, *Combinatorial Optimization: Algorithms and Complexity.* Englewood Cliff, N.J., Prentice Hall.

[17] Przymusinski T., On the declarative and procedural semantics of stratified deductive databases. In Minker, ed., *Foundations of Deductive Databases and Logic Programming*, Morgan-Kaufman, 1988.

[18] K.A. Ross, and Y. Sagiv. Monotonic Aggregation in Deductive Databases. In *Proc. 11th ACM Symp. on Princ. of Database Systems*, pages 127–138, 1992.

[19] S. Sudarshan and R. Ramakrishnan. Aggregation and relevance in deductive databases. In *Proc. of the 17th Conference on Very Large Data Bases*, 1991.

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

[21] Van Gelder A., Foundations of Aggregations in Deductive Databases In *Proc. of the Int. Conf. On Deductive and Object-Oriented databases*, 1993.

[22] Zaniolo C., S. Ceri, C. Faloutsos, V.S. Subrahmanian and R. Zicari, *Advanced Database Systems*, Morgan Kaufmann Publishers, 1997.