Fixed-point semantics and the representation of algorithms on large data

Michel de Rougemont
Laboratoire de Recherche en Informatique,
Université de Paris-Sud,
91405 Orsay, France.
e-mail: mdr@lri

Abstract: In the first part of this paper, we differentiate between two fixed-point semantics that can be used to interpret logic-programs using relations together with functions: on the one hand the fixed-point semantic used in logic-programming [12], where no difference is made between data and logical definitions, and on the other hand the fixed-point semantic used in the theory of inductive definitions [13], where the logical definitions are interpreted relative to the data. We take a logic-program defining a boolean predicate P and show that if we follow the first semantic, P is interpreted as false, and that if we follow the second, P is always true. If we view the logic-program as a set Γ of axioms, then $\Gamma \models_{fin} P$, whereas *not* ($\Gamma \models P$), i.e. P is a logical consequence for *finite* structures of Γ, but not a logical consequence of Γ.

In the second part of the paper, we illustrate this fundamental distinction as we try to represent classical (and hence efficient) algorithms, by logic-programs. We take Shortest-paths algorithms on valued graphs as examples and in particular represent Dijkstra's shortest path algorithm as an inductive definition, under the operational semantic introduced in [7,6].

1 Introduction

In order to extend the current limitations on computability in the context of large data two research directions have been studied.

Either new programming languages are designed in order to deal with databases, or classical database languages such as SQL are extended in order to cope with the growing requirements of computing.

As the data is large, another very important component is the theory of algorithms, when the primary property of algorithms besides their denotation, is their complexity, i.e. the classical time-complexity (space-complexity), measuring the number of steps (the number of memory registers) in the worst-case or average case.

Theoretical studies in the "low-polynomial" time hierarchy find direct applications, if they distinguish algorithms of complexity $O(n)$, $O(n.\log n)$, $O(n^2)$ and $O(n^3)$, as they distinguish on large data between effective and non-effective algorithms, where n is the main parameter measuring the size of the data. The notion of an *effective* algorithm has to be seriously refined when dealing with large databases, as empirical evidence seems to indicate that an ineffective algorithm is one whose complexity property is somewhere between $O(n^2)$ and $O(n^3)$. The barrier to break is not the polynomial time barrier, but the $O(n^2)$ barrier.

In this paper, we show how the theory of inductive definitions allows the *representation* of classical efficient algorithms when working with large data. An inductive definition is compiled, using the operational semantic introduced in [7,6], which provides access to relational data stored on disks through *selections only*. We associate a relative complexity with an inductive definition, as we measure the complexity relative to given operators (specified in the schema) and relative to the selection operator on the data. In the implementation, we approximate the cost of selections as constant by storing the data either as a B-tree with secondary indices as required by the selections we perform, or as Bang data-structures [10], refining the

Grids [14].

We therefore obtain a model of computation where the complexities are relative but can be composed in a constructive way to define an *absolute complexity*. In this model we measure the number of *given* operations on a schema, but often distinguish between the classical complexity and the number of selections on the data (noted A(f(n))). An algorithm is A(n),O(n^2) on a schema if its worst-case complexity is quadratic in n, with a linear number of selections on the data.

An algorithm is usually constructed using other algorithms as *given*, and this is why the primary logical complexity measure has to be a *relative measure*, assuming a unit-cost for the given algorithms. As examples, we consider algorithms for shortest-path problems on valued graphs, and in particular Dijkstra's shortest-path algorithm [9,1]. A valued graph is a ternary schema $\underline{E}$(X,Y,Z), where X and Y range over the domain of the graph, and Z over the positive real numbers. $\underline{E}(a,b,i)$ if there is an edge between point a and point b of cost i. We will give various inductive definition for the query SP(x,y,u) such that $SP(a,b,i)$ if the shortest-path between a and b is of cost i.

In the first part of the paper, we emphasize the various fixed-point semantics that can be taken as foundations for logic-programs. We will show that the fixed-point semantic associated with the least-Herbrand model [12], i.e. the semantic taken in classical logic-programming differs with the fixed-point semantic used in inductive definitions [13], for logic-programs with function symbols. When the logic programs are purely relational these semantics coincide. On the class of graphs with a successor as a partial function on the domain, a minimum element inf and a maximum element sup, we take an inductive definition of the predicate Max(x) such that $Max(a)$ if a is at a maximum distance from inf [1]. We then consider P() $\Leftarrow \exists x Max(x)$. The fixed-point semantic based on the least Herbrand model interprets P() as false, but the fixed-point semantic based on inductive definitions always interprets P() as true.

This phenomenon is fundamental for the representation of algorithms, as some basic constructions used by algorithms may be effective on finite structures, but non-constructive on infinite structures. In actual fact, we show that the query SP on valued graphs has the same property as Max. If we define Q() $\Leftarrow \exists u\, SP(inf, sup, u)$, then Q is always true on finite graphs, but may be false on some infinite graphs. We then give various inductive definitions for SP, that correspond to different algorithm, and in particular to Dijkstra's shortest-path algorithm. This inductive definition is A(n),O(n^2), but breaks the O(n^2) barrier for the average complexity, and allows us to solve the problem on large data.

In the second section, we review the two fixed-point semantics, and exhibit a logic-program that differentiates them. In the third section, we relate the previous phenomenon with the query SP, and make some general remarks concerning the definability of SP in various logic-based languages. We then give two inductive definitions for SP, one of them representing Dijkstra's shortest path algorithm, and make a comparison with other approaches, the approach of "recursive queries" in databases [3], and the classical approach to represent algorithms [1]. In the fourth section we explain the practical side of this approach, as a prototype computing optimum routes on the German railway database is built following this theory.

2 Fixed-point semantics.

2.1 Notations

We assume that data is given as sets of tuples defining relational sets $\underline{R}_1,..,\underline{R}_k$. $\underline{R}_i(a_1,..,a_j)$ iff $<a_1,...,a_j>$ is a tuple of arity j in the set $\underline{R}_i$, where $a_1,...,a_j \in D$, for a finite set D. A *database* is a relational structure DB= $<D, \underline{R}_1,..,\underline{R}_k>$ and a *database schema* is the class **K** of all finite relational structures DB of similar signature. A *logical database* is a logical expansion of a database, i.e. a structure **U**=$<D, \underline{R}_1,..,\underline{R}_k, R_1,..,R_l, f_1,...,f_m, F_1,...,F_p>$, where $R_1,...,R_l$ are relations on D, $f_1,...,f_m$ are functions on D, $F_1,...,F_p$ are functionals [2], A *logical schema* is the class **K** of all finite structures **U** of similar signatures. For a logical database **U**, $\underline{R}_1,...,\underline{R}_k$ are *base relations*, whereas $\underline{R}_1,..,\underline{R}_k$, $R_1,..,R_l$ are *explicit relations*.

The base relations are stored on secondary storage, and are accessed through selections only: if

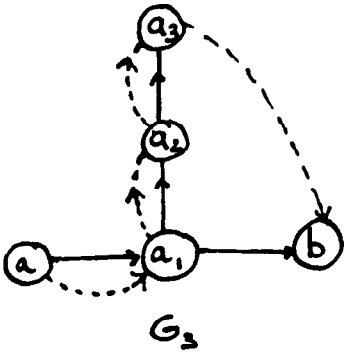
[1] The inductive definition of the predicate Max in [11] is fundamental to observe that inductive queries are closed under complement.

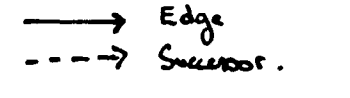
[2] A functional takes a relation, a function or a set as arguments, and returns a value of D. For example the Functional Min, takes a finite set as argument and returns the minimum element in that set. $Min(S) = a$ if $a \in S$, and is the minimum elementof the finite set S.

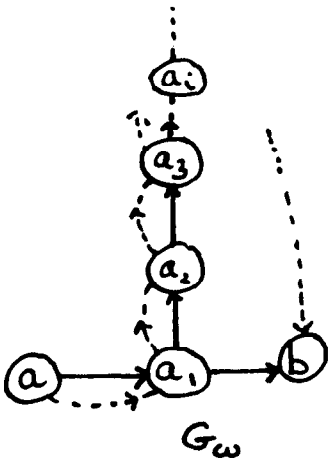
$\underline{R_j}(X_1, ..., X_j)$ is a schema of arity j, and contains n_i tuples, then a selection on an arbitrary set of attributes producing m tuples is done in time AD+$\alpha.log n_i + \beta.m$, where α, β are small in comparison with the constant AD (disk access). In practice, m is small and the cost of a selection can be considered as constant.

The logical schemas that we consider contain an ordering of the domain, i.e. the restriction of the lexicographic ordering to the finite domain D. It is implicetely used by the data structures to ensure that the selections are done in constant time. We assume that a successor function (suc), and a predecessor function (pre) are explicitely given in the logical schemas. A constant function $inf()$ defines the minimum element and another function $sup()$ defines the maximum element of the structure. The predecessor of $inf()$ is undefined ($pre(inf()) \uparrow$), and the successor of $sup()$ is also undefined ($suc(inf()) \uparrow$). As customary, we abbreviate $inf()$ and $sup()$ with inf and sup, treating the constant functions as distinguished elements.

- Example 1: Let K be the class of finite graphs G_n=< $D_n, \underline{E}, succ, pre, inf, sup$ >, D=$\{a, a_1, ..., a_n, b\}$, $\underline{E} \subset D.D$, such that there is an edge between a and a_1 , a_1 and b, and between a_i and a_{i+1} for $i \geq 1$. The successor function starts with $inf()$=a, then joins $a_1, ...a_n$ and then $sup()$=b. The predecesssor function is the inverse of the successor. We represent G_3 and the infinite graph G_ω, where ω={1,2,3,....}.:

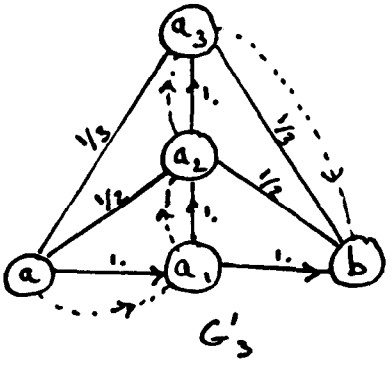


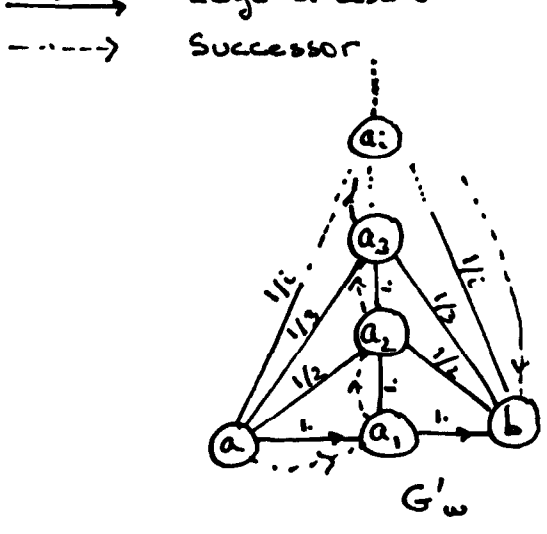




- Example 2: Let K' be the class of structures G'n where each G'n is a valued graph, with the functions $suc, pre, inf, sup.$ as in example 1. G'_n also uses the set of positive real numbers R as parameters, and the function +. We write G'_n =< $D_n, \underline{E}, suc, pre, inf, sup, Min; R, +$ >, where $\underline{E} \subset D.D.R$.

The edges of example 1 are of cost 1., but in addition there are new edges between a and a_i, a_i and b of cost $1/i$ for $i \geq 1$. We represent G' 3 and G'ω





2.2 Inductive queries: data as a finite model.

To a structure **U** of a class **K**, we associate the first-order language L(**K**) with equality: it has first-order variables x,y,z,... ranging over D, relational symbols $\underline{R}_1,..,\underline{R}_k$, $R_1,..,R_l$, the identity symbol =, functions symbols $f_1,...,f_m$ and the usual logical symbols $\vee, \wedge, \exists, \forall, \neg, =, \neq$.

The extended first-order language $L_1(K)$ includes L(K), but in addition *expressions* built with functionals.

- If ψ is a 1-order formula and if Min a functional taking a set as argument, then the expression $\exists u[x = Min(\{u\}) \wedge \psi(x,y,u))]$ is an extended 1-order formula. The interpretation of this formula is:

  $[\exists u(x = Min(\{u\}) \wedge \psi(x,y,u))]^{\mathbf{U}}(a,b)$ iff $S=\{c/[\psi(a,b,c)]^{\mathbf{U}}\}$ and $a = Min(S)$.

  This interpreation is exactly the one taken in languages such as SQL. One computes the set S, and then applies the functional Min. Notice that the notation Min({u}), is strictly equivalent to the "GROUP by u " notation in SQL.

Let L(**K**) be the extended first-order language, with relational symbols $R_1,..,R_k$, first-order variables x_1, x_2,..., and the classical functions and Functionals. Assume the classical notions of satisfiability for formulas $F(x_1,...,x_k, S_1,...,S_l)$, the notion of S *occuring positively in F* [13], and the notion of a *relational query* defined in [4,5,2].

Definition [13]: An inductive system with parameters on a class **K** is a sequence of formulas $F_1,...,F_k$ in the language $L(\mathbf{K})\cup\{S_1,...,S_k\}$, such that each S_i occurs positively in each F_j for $1 \leq i,j \leq k$, and such that each S_i is of arity r_i+p. We write a system as:

$$\begin{cases} S_1(x_{11}...x_{1r_1}, y_1.....y_p) \Leftarrow F_1(x_{11}...x_{1r_1}, S_1...S_k : y_1...y_p) \\ S_2(x_{21}...x_{2r_1}, y_1.....y_p) \Leftarrow F_2(x_{21}...x_{2r_2}, S_1...S_k : y_1...y_p) \\ \quad .... \quad ... \\ \quad ... \quad ... \\ S_k(x_{k1}...x_{kr_1}, y_1.....y_p) \Leftarrow F_k(x_{k1}...x_{kr_k}, S_1...S_k : y_1...y_p) \end{cases}$$

The parameters $y_1,...,y_p$ are kept constant in the recursions, whereas the x_{ij}'s play the role of recursion variables. The *fixed-point semantic* [13] associates with the system S and with each structure **U**, the *fixed-points* $[S_i^\infty]^{\mathbf{U}}$ defined at the *finite* closure ordinal λ, for the stages defined as: $[S_i^0]^{\mathbf{U}} = \phi$ (the empty set); $[S_i^{i+1}]^{\mathbf{U}} = [F_i([S_1^i]^{\mathbf{U}},...., [S_m^i]^{\mathbf{U}})]^{\mathbf{U}}$.

Then $[S_i^\infty]^{\mathbf{U}} = [S_i^{\lambda+1}]^{\mathbf{U}} = [S_i^\lambda]^{\mathbf{U}}$.

If the formulas F_i contain some functionals (Min, Max, etc....) then the iterations are not necessarily monotone: in this case the relation $[S_i^\infty]^{\mathbf{U}}$ is defined as the *cumulative fixed-point*, i.e. the limit of the sequence: $[S_i^0]^{\mathbf{U}}$, $[S_i^1]^{\mathbf{U}}$,..... $[S_i^j]^{\mathbf{U}}$, $[S_i^{j+1}]^{\mathbf{U}}$,....where $[S_i^{j+1}]^{\mathbf{U}} = [S_i^j]^{\mathbf{U}} \cup [F_i([S_1^j]^{\mathbf{U}},...., [S_m^j]^{\mathbf{U}})]^{\mathbf{U}}$.

Definition:[8] A relational query is inductive of *dimension d* on a class **K** if there exists a system of dimension d such that for all **U**: $[Q]^{\mathbf{U}} = [S_1^\infty]^{\mathbf{U}}$.

- Consider the example 1: we can define the following queries:

$$\{\text{Anc(x,y)} \Leftarrow \underline{E}(x,y) \vee \exists z[\underline{E}(x,z) \wedge Anc(z,y)]$$

$$\begin{cases} \text{Con()} \Leftarrow \forall x Comp(x). \\ \text{Comp(x)} \Leftarrow \underline{E}(\inf, x) \vee \exists z[Comp(z) \wedge \underline{E}(z,x)]. \end{cases}$$

  The first system defines the classical Anc query, with an existential induction of dimension 1. The second system defines the boolean query (true or false) Con(): Con() *if there is a path from inf to* all other points. Con() is inductive of dimension 1, but *not* existential, as a universal predicate appears in the first formula of the system.

- Consider the class of valued graphs as in example 2, with the functional Min2 that takes 2 sets as arguments and returns the minimum element of both sets.

  $Arcmin(x,y,u) \Leftarrow \underline{E}(x,y,v), u = Min(\{v\})$.

  We verify that $Arcmin(a,b,i)$ if i is the minimum of $\{j \ / \ \underline{E}(a,b,j)\}$.

$$\begin{cases} SP(x,y,u) \Leftarrow \exists v\, Ancm(x,v,y) \wedge u = Min(\{v\}). \\ Ancm(x,u,y) \Leftarrow \underline{E}(x,y,u) \vee \exists z,v,w[\underline{E}(x,z,v) \wedge \\ \qquad Ancm(z,w,y) \wedge t = v + w \wedge \\ \qquad Ancm(x,j,y) \wedge u = Min2(\{t\},\{j\})]. \end{cases}$$

This second system defines SP, of dimension 2 using y as a parameter, by induction on the length of the paths. $Ancm(a,i,b)$ if there is a path of length i obtained by taking the path of minimum cost among all the minimum paths of length i-1. This induction is non-monotonic, as it uses the functional Min2.

We say that a boolean query Q is *always true on a class K*, if it is true for all finite structures of K.

2.3 The least Herbrand model.

In this classical approach to logic programming [12], the definitions are viewed as first-order axioms, i.e. replacing $\Leftarrow$ by the logical implication $\leftarrow$. The relational data are considered as first-order axioms, together with the relational data.

What we viewed as a set of definitions, is now viewed as a set of clauses. In case of existential inductions, which are positive (no negation on the given explicit relations), the clauses are Horn-clauses, built from terms containing possibly some functions symbols.

Within this framework, the set of terms of a program P (a set of clauses) is the Herbrand Universe, defining the Herbrand interpretations. The interesting one is the least Herbrand model that can be defined by iterating a monotone operator T_P, defining the set of clauses on the left hand side of $\leftarrow$, given the set of clauses on the right hand side (see [12]).

For a boolean predicate P() is true if P() $\in T_P \uparrow \omega$, following LLoyd's notations. In the case of logic-programs without function symbols, the two interpretations are clearly equivalent:

Proposition: For logic-programs without functions symbols, the fixed-point semantics based on inductive defintion and on the least Herbrand model coincide:

Proof: By induction on the stages, it is simple to realize that a boolean predicate P is such that for all U, $[P^i]^U$ iff P() $\in T_P^i$, i.e. the i-th iteration of T_P $\dashv$.

In this case, the Herbrand model and the finite structure are isomorphic. In the more interesting case of logic-programs with functions symbols, the Herbrand base is infinite, whereas the structure is finite, an entirely different situation.

2.4 A logic-program for the Maximum.

In this section we present a logic-program with functions (*pre* and *succ*) that distinguishes the two fixed-point semantics. The program is best understood as an inductive definition of the predicate Max(x), on the class of graphs of example 1. We first show an inductive definition of Max and of the boolean predicate P() saying that there exists a maximum. We then transform the inductive definition into an existential positive one, making an extensive use of the functions. At this point we reach a unique logic-program that can be interpreted following the two semantics we introduced. P() is true for all G_n of example 1, following the fixed-point semantic of the inductive definitions, but P() is not a logical consequence of the axioms, as the model G_ω is such that P() is false.

2.4.1 definition of P and Max.

Consider the system defining the boolean P on the class of graphs of example 1 with the relation $\leq$, the ordering on the domain: in a first step we define *Ancm*, and then define P on the new class expanded with *Ancm*. We then transform this induction into an existential induction.

$$\begin{cases} P() \Leftarrow \exists x\, Max(x). \\ \mathrm{Max}(x) \Leftarrow \exists u\, Ancm(x,u) \wedge [\forall y(\exists v\, Ancm(y,v) \wedge y \neq x \rightarrow v \leq u)]. \\ Ancm(x,u) \Leftarrow (x = inf \wedge u = inf) \vee (\exists y,v\, Ancm(y,v) \wedge \underline{E}(y,x), u = succ(v)). \\ \underline{E}(x,y) \Leftarrow (x = inf \wedge y = succ(inf)) \vee (x = succ(inf) \wedge y = sup) \vee (\exists u,v \underline{E}(u,v) \wedge x = v \wedge y = succ(v) \wedge y \neq sup). \end{cases}$$

We simply state that $Ancm(a,i)$ if a is at a distance i from inf, and that $Max(a)$ if $Ancm(a,j)$ and for all c different from a if $Ancm(c,i)$ then $i \leq j$. The definition of E, axiomatizes the class of graphs of the

figure 1. The induction defining *Ancm* is positive and existential, whereas the one defining P is universal and negative in *Ancm*. We can however always replace this two steps system with a one step existential system, using the functions, and implicitely the *finiteness of the structures*. We then obtain:

$$\begin{cases} P() \Leftarrow \exists x Max(x). \\ Max(x) \Leftarrow \exists u Ancm(x,u) \wedge [Checkmax(x,u)]. \\ Checkmax(x,u) \Leftarrow Checkrec(u,sup). \\ Checkrec(u,y) \Leftarrow y = inf \vee (\exists v Ancm(y,v) \wedge v < y \wedge Checkrec(u,pre(y)). \\ Ancm(x,u) \Leftarrow (x = inf \wedge u = inf) \vee (\exists y,v Ancm(y,v) \wedge \underline{E}(y,x), u = succ(v)). \\ \underline{E}(x,y) \Leftarrow (x = inf \wedge y = succ(inf)) \vee (x = succ(inf) \wedge y = sup) \vee (\exists u,v \underline{E}(u,v) \wedge x = v \wedge y = succ(v) \wedge y \neq sup). \end{cases}$$

In this new induction, we replaced the universal quantifier by a recursive predicate (Checkrec), verifying that all points are at a distance less than the distance of the maximum point (doing a linear search on the domain). The program we obtain is an existential induction, hence can be viewed as a set of clauses with function symbols.

Proposition: P() is always true on the class K of example 1.

Proof: One verifies that the definition of Max is correct, and we know that any finite graph has a maximum. ⊣

Proposition: As a logic-program P() is false.

Proof: P() is true iff P $\in T_P \uparrow \omega$ iff P is a logical consequence of the axiom. But P is not a logical consequence of the axiom, as for the model G_ω, P is false, as there is no maximum. Hence P() is false using the logic-programming semantic. ⊣

2.5 The logical foundations of inductive queries

The foundations of classical logic-programming is that a predicate is true iff it is a logical consequence of the axioms. The previous example shows that the semantic of inductive queries, because it differs with the logic-programming semantic, will not share this fundamental property. There is however another property that can be taken as logical foundation:

Theorem: For a set Γ of first-order axioms, and a predicate P appearing on the left-hand side, P() is always true as an inductive query iff $\Gamma \models_{fin} P()$.

Proof: For any finite structure P() is true, and we can then build a constructive proof of P, by computing P. This proof will show that P is a logical consequence for finite structures of the logic-program, viewed as a set Γ of first-order axioms. ⊣.

This simple result, generalizes however to some second-order axioms. We will see such examples later.

3 The representation of algorithms.

The previous phenomenon is important however, if one tries to represent classical algorithms as logic-programs, because the basic constructions of an algorithm may use the finiteness of the data implicitely. Notice that classical query languages also use the finiteness of the data implicitely: a selection of a relation yields *effectively* a relation, because the domain is finite. On an infinite structure, a selection is ineffective. In the same way, an inductive definition is effective on finite structures, but ineffective on an infinite one.

If a proper distinction between recursion variables and parameters is made, we can then take the operational semantic introduced in [7,6], that can be understood as: pass the parameters *by value*, then iterate the set constructions until the closure ordinal is reached. With this semantic some very efficient algorithm can be represented as inductive definitions.

3.1 The model.

In this paper we have been using an important expression: *A logic-program represents an algorithm.* We

now define this concept precisely: Let P be a program having a certain relative complexity property, and let A be an algorithm having another relative complexity, i.e. assuming a unit-cost for the given operations.

Definition: A program P *represents* an algorithm A, if they have the same denotation and the same relative complexity properties, within a constant factor.

As an example, we concentrate on algorithms defining SP on the class of valued graphs, as in the example 2. This is an example of both practical and theoretic interest. In practice, we don't look for an arbitrary path (as in Anc for example), but for those paths that satisfy a condition (as in SP). The theoretical interest in SP in explained next.

3.2 Definability of SP.

Consider the class of graphs as in example 2, and let us define the boolean predicate Q() as follows: $Q() \Leftarrow \exists u SP(inf, sup, u)$.

Proposition: Q is always true on the class K' of example 2, but false on G'_ω.

*Proof:*We can repeat the same argument as in the case of the Maximum: every finite G' has a minimum path from *inf*=a to *sup*=b. The infinite graph G'_ω does not, as given any path, there always exists a shorter one. ⊣

In addition we can observe that SP is not definable without some basic non-monotone constructions. In particular it is not definable in Datalog.

It is essential to observe the monotonicity of queries defined in Datalog. If Q is an existentially inductive query and if Q is true on a structure **U** (database state), then if we add some tuples to **U**, defining **U'**, then Q will be true in **U'**. This relects the fact that an existential proof is maintained in larger structures, or the fact that a calculus for first-order logic must be monotone.

Proposition: SP is not definable in Datalog.

Proof: suppose it were. On G'_3, we would have $G'_3 \models SP(inf, sup, 2/3)$. If we now add the extra relational edges to define G'_4 (3 edges more), then $not(G'_4 \models SP(inf, sup, 2/3)$, as the minimum path is of cost 1/2. We would then contradict the monotonicity. Hence SP is not Datalog definable. ⊣

3.3 An inductive definition of SP

We now switch to the classical logic-programming notations, where non-free variables in a definition are by default existentially quantified, where "," replaces ∧, ";" replaces ∨, and ":" replaces ⇐. We do not use the symbol ":-", replacing ←, because we now adopt the fixed-point semantic of inductive queries. In fact we now consider systems with the functionals Min, and Min2, under the cumulative fixed-point semantic (see section 2). We take the Φ [7,6] notations, where the defined predicate of a system is emphasized, following the key-word "ind". This induction reflects the one we gave as an example in section 2, and is on the length of the paths.

```
ind SP(x,y,u)

SP(x,y,u) : Ancm(x,v,y),u=Min({v}).

Ancm(x,u,y):E(x,y,u) ;[E(x,z,v),Ancm(z,w,y),
      t=v+w,Ancm(x,j,y), u=Min2({t},{j})].
```

. .

This induction is of dimension 2, and uses y as a parameter. Its relative complexity is $O(n^2)$, but its absolute complexity is worse than $O(n^3)$, and therefore is not very interesting.

3.4 Dijkstra's shortest path algorithm.

Dijkstra's algorithm is captured by a more complex induction than the one described in the previous example. The essential of the definition is the simultaneous induction of Ancm, together with a binary relation S(x,y).

For y=*a* let $S^i = \{ u / S^i(u, a) \}$, i.e. the set defined at stage i. The essential and non-trivial property of the system [Ancm,S] is the fact that for all i, $Ancm^i$ contains the shortest path relative to the subgraph defined by S^i. This induction is non-monotonic, as it uses the functionals Min2 and Min, but also an explicit negation. We now take the full class of non-motonone inductions, under the cumulative fixed-point semantic.

The compilation techniques used in [7,6] generalizes to this case.

```
ind SP(x,y,u)

SP(x,y,u) : Ancm(x,v,y),u=Min({v}).

Ancm(x,v ,y) : E(x,u,z),S(u,y),Ancm(u,t,y),
               Ancm(x,w,y), v=Min2({w},{t+z})

             ; E(x,v,y).

S(u,y) : Ancm(u,z',y),Ancm(v,z,y),
         not S(u,y),z'=Min({z}).

..
```

This system is of dimension 2, where y is a parameter. The definition of Ancm can be understood as: *The marking at stage i for y=a, is obtained from the marking at stage i-1, by considering the new paths from the new b such that* S^{i-1}(b,a), *and by taking their minimum.* The definition of S can be understood as : *The new point b at stage i is obtained by taking the point of minimum cost among the points that are not in* S^{i-1}. What is fundamental to observe is that, we only explore the new paths from *one point b*, computed by S. This makes the program far more efficient than the one based on the induction on the length of the paths, where all the paths of length k are generated from some paths of length k-1.

The principle of this induction is exactly the principle of *Dijkstra shortest path algorithm* [9,1]. It shows the strong relationship that exists between a specific algorithm and an inductive definition. If one looks at a book on algorithms [1] and compares the representation of this algorithm with the one we gave, one can observe that the graph is usually represented as an array, and that the complexity analysis of the algorithm assumes that accessing an element in the array can be done in 1 step. This hypothesis is true for n small, but false for large n, in contradiction with an asymptotic complexity analysis. In this representation, the program is compiled in such a way that we only access the data with a selection (in fact a selection on the first attribute). The hypothesis of a selection in constant time is true, the overall complexity analysis can be done by the compiler, and *is compatible with n large*. The program takes 2 recursion variables, and hence is of worst-case complexity $O(n^2)$, but in fact is far better for the average complexity. This requires a finer analysis of the induction, in order to show that only *one point* is explored, as the inductive computations proceed.

3.5 Comparison with other approaches

3.5.1 Recursive queries in databases

It is important to note that the operational semantics presented in [3], although presented in terms of classical logic-programming terms, seem to adopt the fixed-point semantic of inductive queries, as they lead to computations of finite sets, implicitely using at some point the finiteness of the data. Most of these methods concentrate on Datalog, which does not allow the definition of SP, as we have seen using a simple monotonicity argument.

The use of these operational semantic to represent algorithms is very unclear, as is the promised *optimization* that these methods claim.

3.5.2 The classical representation of algorithms

The logical representation of a classical algorithm has to be compared with the one given in classical books on algorithms [1]. In these books, the data is usually stored in an array, and one assumes that accessing an element of the array is done in constant time. This hypothesis is however only true for small graphs, and false for large graphs. Hence the complexity analysis, as an asymptotic measure, i.e. for large graphs is in contradiction with the implicit hypothesis on the relative costs. What we obtain in our representation as a logical definition, is a representation of the algorithm, where the basic hypothesis are compatible with n large, and hence the complexity analysis makes sense. What is however fundamental to observe is that the relative complexity of the program is syntactically captured by the dimension of the inductive definition.

4 The practical use

We have been applying the previous theory in designing a practical system to compute optimum routes

from the German railway network database. The schema is far more complex than the valued graphs we considered, as it gives the precise schedule of trains (10 attributes). If we look for the shortest route in time between two stations, we proceed as follows. In a first step, we write Dijkstra's algorithm as an inductive definition with the time, as a cost function. It is interesting to note, that selections on only one attribute are performed (departing station), and that a B-tree structure without secondary indices suffices to store the data. When compiled SP executes on a simulated database within 20 to 60 seconds on a SUN3. and this is far too long for real applications.

We precompute all shortest paths between the 50 largest German cities, using the previous definition of SP. We then code a natural heuristics SP1 for SP than can be understood as: to compute the shortest route between a and b, if a and b are large cities then look at the precomputed routes, otherwise determine two large cities $a1$ and $b1$ that can be reached from a and b by direct trains in time ta and tb; let $t1$ be the time of the shortest route between $a1$ and $b1$; then the time of the shortest route between a and b is $t1 + ta + tb$.

This heuristics for SP1 can then be defined by a very simple inductive definition on the expanded schema, and is compiled using the same techniques. It executes within 10 seconds, and with a high probability gives a solution to the problem.

References

[1] A. Aho, J. Hopcroft, and J. Ullman. *The design and analysis of computer algorithms.* Addison-Wesley, 1974.

[2] Ullman J. Aho A. Universality of data retrieval languages. In *Principles of Programming languages*, pages 11–,117, ACM, 1979.

[3] F. Bancilhon and R. Ramakrishnan. An amateur's introduction to recursive query processing strategies. *SIGMOD*, 1986.

[4] J. Barwise and Y. Moschovakis. Global inductive definability. *Journal of Symbolic Logic*, 43(3):521–534, 1978.

[5] A. Chandra and D. Harel. Structure and complexity of relational queries. *Journal of Computer and System Sciences*, 25(1):99–128, 1982.

[6] M. de Rougemont. Calculabilite et bases de donnees. *Techniques et Science informatiques*, 6(5):419–434, 1987.

[7] M. de Rougemont. The computation of inductive queries by machines. In *Logic, Language and Computation*, ASL, 1985. Abstract in *Journal of Symbolic Logic 51(3):p. 839, 1986.*

[8] M. de Rougemont. Second-order and inductive definability on finite structures. *Zeitschrift fur Mathematische Logik und Grundlagen der Mathematik*, 33(1):47–63, 1987.

[9] E. Dijkstra. A note on two problems in connexion with graphs. *Numerische mathematik*, 1:269–271, 1959.

[10] M. Freeston. The bang file: a new kind of grid file. In *SIGMOD*, 1987.

[11] N. Immerman. Relational queries computable in polynomial time. In *Symposium on the theory of Computing*, pages 147–152, ACM, 1982.

[12] J.W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, 1984.

[13] Y. Moschovakis. *Elementary Induction on Abstract Structures.* North-Holland, 1974.

[14] J. Nievergelt, H. Hinterberger, and K.C. Sevcik. The grid file: an adaptable, symmetric multikey file structure. *ACM Transactions on Database Systems*, 9(1):38–71, 1984.