Logical Algorithms

Harald Ganzinger[1] and David McAllester[2]

[1] MPI Informatik, D-66123 Saarbrücken, Germany, `hg@mpi-sb.mpg.de`
[2] AT&T Labs-Research

Abstract. It is widely accepted that many algorithms can be concisely and clearly expressed as logical inference rules. However, logic programming has been inappropriate for the study of the running time of algorithms because there has not been a clear and precise model of the run time of a logic program. We present a logic programming model of computation appropriate for the study of the run time of a wide variety of algorithms.

1 Introduction

It is widely accepted that many algorithms can be concisely and clearly expressed as logical inference rules. Unfortunately there is a fundamental problem in using inference rules in the study of algorithms — the precise running time of inference rules can depend on implementation details such as indexing. Run time is one of the most fundamental properties of any algorithm. Textbooks on algorithms and data structures begin by giving a model of computation — usually a random access memory (RAM) machine — which formally defines the notion of run time. Algorithms must be presented in a way that makes their execution on the underlying model sufficiently clear that precise run time analysis is possible. We will use the phrase *algorithmic* model of computation to mean a model of computation that includes a well defined notion of run time. Here we present a new algorithmic logic programming model of computation. We prove that the abstract notion of run time in this model is implementable on a RAM machine extended with constant time hash table operations. We also show how a variety of algorithms that are not easily expressed in earlier algorithmic logic programming models are naturally expressed in our new model.

McAllester recently introduced a pure logic programming algorithmic model. In this model a program is simply a finite set of first order Horn clauses [5]. The program (set of rules) R defines a mapping from an input database D to an output database $R(D)$ where $R(D)$ is all assertions derivable from the assertions in D using the rules in R. The running time of R on input D is defined to be the number of assertions in $R(D)$ plus the number of "prefix firings" in the computation. A prefix firing is an instance of a prefix of the antecedents of a rule such that the prefix instance is true in the final closure $R(D)$ (see [5] for details). McAllester proved that an interpreter can be constructed for pure logic programs such that the program can be executed on a RAM machine extended with constant time hash table operations in time proportional to its abstract run time.

A variety of efficient algorithms can be naturally expressed as pure logic programs, especially dynamic programming algorithms in parsing and in the static analysis of computer programs. However, many other problems appear not to have efficient solutions as pure programs. For example, pure logic programs do not seem to support an efficient union-find algorithm. In the same paper, McAllester also presents a more elaborate logic-programming model that incorporates union-find specifically into the model. More recently Ganzinger and McAllester [2] gave a more general algorithmic logic programming model, one including rule priorities and deletion, which allows union-find to be implemented efficiently, as well as many other algorithms not naturally covered by the pure logic programming model.

While rule priorities and deletions greatly expand the power of the model, they still fail to cover certain algorithms. Dijkstra shortest path, for example, still appears to not have a natural representation. The contribution of this paper is an algorithmic logic programming model with yet broader coverage. In particular we allow different instances of the same rule to run at different priorities. We prove that an interpreter (or compiler) can be constructed on a RAM machine extended with constant time hash table operations such that the time taken to run a program is proportional to the abstract notion of run time. As evidence for the power of our model we demonstrate implementations of Dijkstra's shortest path and of minimal spanning tree.

If one ignores the abstract notion of run time, our logic programming model is not particularly novel. The language is variant of bottom-up logic programming. Bottom-up logic programming has been widely studied in the context of deductive databases [13, 11, 6, 12]. Bottom-up logic programming is closely related to "memoing" or "tabling" for Prolog programs [10, 9, 1]. The bottom-up language described here allows deletion. Our notion of deletion is superficially similar to widely studied notions of negation in logic programming such as well founded (stratified) programs [8, 4] and stable-model semantics [3]. Here, however, we use a "don't-care" nondeterministic semantics which does not require the program to be well founded and where the final database need not be stable. Deletion in logic programming has also been modeled with linear logic [7]. The linear logic approach is similar to ours but it has not been given an abstract notion of run time executable on a RAM machine. Our notion of rule priorities is similar to notions previously studied in the logic programming literature, e.g., [14]. But to our knowledge earlier work has not provided an abstract notion of run time implementable on a RAM machine.

2 A Broad Algorithmic Logic Programming Model

To support a wide variety of algorithms it is necessary to support arithmetic. To support arithmetic we define atomic formulas and terms with the following grammar.

$$\begin{aligned}
\Phi &\equiv P(\tau_1,\ \ldots,\ \tau_k)\ \mid\ N_1 < N_2\ \mid\ N_1 \leq N_2 \\
\tau &\equiv H\ \mid\ N \\
H &\equiv x\ \mid\ f(\tau_1,\ \ldots,\ \tau_k) \\
N &\equiv i\ \mid\ n\ \mid\ N_1 + N_2\ \mid\ N_1 * N_2
\end{aligned}$$

This is two-sorted grammar with a sort for Herbrand terms (H) and a separate sort for integers (N). We allow predicates and functions to take either sort as arguments although we assume that each predicate and function symbol has a specified arity and a specified sort for each argument. In the expression $P(\tau_1,\ \ldots,\ \tau_k)$ we require that P be a predicate symbol of arity (number of arguments) k and that each argument is of the appropriate sort for the predicate P. A similar comment applies to the expression $f(\tau_1,\ \ldots,\ \tau_k)$ in the grammar for H. The function symbol f in the grammar for Herbrand terms should be viewed as a data constructor. The precise semantics is defined below. Note that there are two sorts of variables — Herbrand variables such as x in the grammar for Herbrand terms and integer variables such as i in the grammar for integer terms. In the grammar for integer terms, n ranges over any integer constant.

A *ground term* is a term not containing variables or arithmetic operations. A numerical constant such as 5 is a ground term. A constant symbol such as c is also a ground term where we view c as a function of no arguments. The term $f(c, 4)$ is also a valid ground term provided that f is a function of two arguments, the first of sort Herbrand term and the second of sort integer. The term $f(c, 2{+}3)$ is not a valid ground term because it contains an arithmetic operation but the term $f(c, 5)$ is. A *ground substitution* is a mapping from variables to ground terms where the value of a Herbrand variable must be a ground term of the form $f(\tau_1,\ \cdots,\ \tau_k)$ and the value of an integer variable must be an integer constant. If τ is a term (as defined by the above grammar) and σ is a ground substitution, then we define $\sigma(\tau)$ to be the result of replacing each variable by its value in σ and then computing the result of all arithmetic operations. For example, if σ maps x to c and i to 3, then $\sigma(f(x,\ i + 4))$ is the ground term $f(c, 7)$. A ground atom is either one of the constants T or F (representing "true" and " false" respectively) or an expression of the form $P(\tau_1,\ \ldots,\ \tau_k)$ where each τ_i is a ground term. If Φ is an atomic formula as defined by the above grammar, and σ is a ground substitution, then we define $\sigma(\Phi)$ to be the result of replacing each variable by its value and then computing arithmetic operations. For example, if σ maps i to 5 then $\sigma(P(c, i))$ is the ground atom $P(c, 5)$ while $\sigma(i < 7)$ is the constant T and $\sigma(7 < i)$ is the constant F.

Here we are interested in inference rules (logic programs) that support deletion. An expression of the form del(A) where A is an atomic formula (as defined by the above grammar) will be called a *deletion assertion*. We use del(A) rather than $\neg A$ to avoid any confusion with the semantic notion of logical negation. We will use the term *assertion* to mean either a ground atom or a deletion assertion. A deletion assertion del(A) is called *ground* if A is a ground atom. We define an *inference rule* to be an expression of the form $A_1 \wedge \ldots \wedge A_n \Rightarrow C$ where each *antecedent* A_i is an assertion (atoms or deletion assertions) not involving addition or multiplication and the *conclusion* C is a finite set of assertions (atoms or deletion assertions) such that every variable in C occurs in some antecedent and all variables in a comparison antecedent, i.e., an antecedent involving $<$ or $\leq$, must occur in some earlier antecedent. A *state* of the computation is a finite set of ground assertions. In the computational processes defined here, states grow monotonically over time — the deletion of A is modeled by the addition of $\mathrm{del}(A)$. An atom A is said to be *visible* in state S if A is the constant T or A is a ground atom such that $A \in S$ and $\mathrm{del}(A) \notin S$, or A is a deletion assertion and $A \in S$.

Deletion makes the behavior of rule sets nondeterministic. For example, consider the following rules with deletion.

$$p \Rightarrow q \qquad q \Rightarrow s, \mathrm{del}(q) \qquad q \Rightarrow w, \mathrm{del}(q)$$

Suppose the initial database contains only p. The first rule fires adding the assertion q. Now either the second or third rule can fire, but once one of them fires the other is blocked because q is no longer visible. Hence the final state is nondeterministically either $\{p,\ q,\ \mathrm{del}(q),\ s\}$ or $\{p,\ q,\ \mathrm{del}(q),\ w\}$. When viewing rules with deletions as algorithms, this nondeterminism is viewed as "don't care" nondeterminism — choices between equal priority rule invocations are made arbitrarily and are not backtracked (priorities are discussed below). As another example consider the following rules.

$$p \Rightarrow q \qquad q \Rightarrow \mathrm{del}(q), w \qquad w \Rightarrow \mathrm{del}(w), q$$

A naive interpretation of deletion might lead one to think that starting from the state $\{p\}$ we get an infinite loop switching between a state where q is visible and a state where w is visible. But under the semantics used here, once an atom is deleted it remains invisible forever. Running the above rules from the start state $\{p\}$ terminates with the state $\{p,\ q,\ \mathrm{del}(q),\ w,\ \mathrm{del}(w)\}$. Our notion of deletion is quite different from semantic negation.

Priorities allow fine grain control for rules with deletion. We define a *program* to be a pair $\langle R,\ \pi\rangle$ where R is a finite set of rules (as defined above) and π maps each rule to an arithmetic expression constructed from integer variables occurring in the first antecedent of the rule. We use the convention that smaller integers represent higher priorities. For $r \in R$, and σ a ground substitution interpreting at least the variables of the first antecedent of r, we let $\pi(r,\ \sigma)$ be the maximum of 1 and the integer $\sigma(\pi(r))$. Note that "priority 1" is the highest possible priority. We say that r has *fixed priority* if $\pi(r)$ is an integer constant, and otherwise we say that r has *variable priority*.

Now consider a fixed program $\langle R,\ \pi\rangle$. An *instance* of r is a pair $\langle r,\ \sigma\rangle$ where σ is a ground substitution defined on (only) the variables occurring in the antecedents of r (and hence also defined on the variables in the conclusion). We will say that the antecedents of the instance $\langle r,\ \sigma\rangle$ *hold* in a state S if, for each antecedent A_i of r, we have that $\sigma(A_i)$ holds in S. We say that an instance $\langle r,\ \sigma\rangle$ is *pending* at state S if $r \in R$; the antecedents of the instance hold in S; and

$S \neq S \cup \sigma(C)$ where C is the conclusion of r. We define the *priority of a state* S, written $\pi(S)$, to be the priority of the highest priority pending rule instance, i.e. $\pi(r, \sigma)$ for some pending instance $\langle r, \sigma\rangle$ such that there is no pending instance $\langle r', \sigma'\rangle$ with $\pi(r', \sigma') < \pi(r, \sigma)$. If there is no pending rule instance for S then S is called *saturated* and we define $\pi(S)$ to be (positive) infinity. We write $S \overset{R,\,\pi}{\to} S'$ if there exists some instance $\langle r, \sigma\rangle$ pending in S with $\pi(r,\sigma) = \pi(S)$ and where $S' = S \cup \sigma(C)$ where C is the conclusion of r. An $\langle R, \pi\rangle$-computation is a sequence $S_0, S_1, \ldots, S_t$ where $S_i \overset{R,\pi}{\to} S_{i+1}$. We also allow infinite computations of the form $S_0, S_1, S_2, \ldots$. A program $\langle R, \pi\rangle$ is said to terminate provided that there are no infinite $\langle R, \pi\rangle$-computations. An $\langle R, \pi\rangle$-computation is called *complete* if it is finite and the final state is saturated. If there exists a complete $\langle R, \pi\rangle$-computation from S_0 to S_t then S_t is called an *$\langle R, \pi\rangle$-saturation* of S_0.

We now define an abstract notion of running time for terminating programs. Consider a rule $r \in R$. If r has n antecedents then for $1 \leq j \leq n$ we let r_j denote the jth antecedent of r. A *prefix instance* of r is a triple $\langle r, i, \sigma\rangle$ where $1 \leq i \leq n$ and σ is a ground substitution defined on (only) the variables occurring in the first i antecedents $r_1, \ldots, r_i$. An instance of r as defined earlier is just a prefix instance with $i = n$.

Definition of Abstract Running Time: Consider a complete finite computation C starting in state S_0 and ending in state S_t. A *weak prefix firing* of r in C is a prefix instance $\langle r, i, \sigma\rangle$ of r such that for $1 \leq j \leq i$ we have that either $\sigma(r_j)$ is T or $\sigma(r_j) \in S_t$ (note that elements of S_t need not hold in S_t — S_t may contain both A and $\mathrm{del}(A)$). A *strong prefix firing* of r is a weak prefix firing $\langle r, i, \sigma\rangle$ such that there exists a state S with $\pi(S) \geq \pi(r, \sigma)$ such that all the antecedents of the firing hold in S, i.e., for all $1 \leq j \leq i$ we have that $\sigma(r_j)$ holds in S. An *antecedent instance* of a rule r is an element of S_t that is an instance of an antecedent of r. The abstract running time of the computation is defined to be $|S_0| + P_f + (P_v + A_v)\log N$ where $|S_0|$ is the number of assertions in S_0, P_f is the number of strong prefix firings of fixed priority rules; P_v is the number of strong prefix firings of variable priority rules; A_v is the number of antecedent instances of variable priority rules; and N is the number of distinct priorities, i.e., the number of priorities of the form $\pi(r,\sigma)$ with $\sigma(r_1) \in S_t$.

Without deletion there is no distinction between week and strong prefix firings — any weak prefix firing is strong by virtue of the final state S_t which as infinite priority. Also note that an assertion which is asserted and then deleted, both at priority higher than the priority of a rule r, does not participate in strong prefix firings of r. However, if such an assertion matches an antecedent of r then it counts as an antecedent instance no matter how quickly it is deleted. As a simple case consider the rule $P(x,y) \land P(y,z) \land Q(y) \Rightarrow P(x,z)$ with fixed priority 1 and assume there is no deletion. Any firing of the first two antecedents corresponds to at most one firing of all three antecedents but many firings of the first two antecedents may fail to correspond to any firing of all three. Note that the (logically equivalent) rule $P(x, y) \wedge Q(y) \wedge P(y, z) \Rightarrow P(x, z)$ is more efficient, i.e., has fewer prefix firings. We rely on the programmer to write rules in an efficient form given an understanding of the abstract notion of run time. The main result of this paper is the following which extends the result in [2] beyond fixed priority rules.

Theorem 1. *For any terminating program $\langle R,\ \pi\rangle$ there exists an algorithm running on a RAM machine extended with constant time hash table operations such that for any given initial state S_0 the algorithm computes a complete $\langle R,\ \pi\rangle$-computation from S_0 in time proportional to the abstract running time of the generated computation.*

3 Examples

Our first example is a simple algorithm for determining whether a given input graph is bipartite. A graph is bipartite if its nodes can be partitioned into two subsets A and B such that edges do not connect any pair of nodes of the same subset. The rules in figure 1 determine bipartiteness — the graph is bipartite unless these rules assign both labels to the same node. Each rule is labeled with a name and a priority. In this example all rules have fixed priority. Furthermore, all rules have priority 1 except rule (B6) which has (lower) priority 2. Note that in any state with priority 2 or greater we have that $\mathsf{unlabeled}(u)$ is visible if and only if there is no assertion of the form $\mathsf{labeled}(u,\ k)$. Since all rules in the example have fixed priority, the abstract running time is just the size of the input data base plus the number of strong prefix firings of the rules. One can check that every rule has at most $O(e)$ (weak) prefix firings where e is the number of edges in the input graph. Hence the abstract running time of the algorithm is linear in the size of the input graph.

The Dijkstra shortest path algorithm is depicted in Figure 2 where the assertion $E(u, c, v)$ represents a directed edge from u to v with integer distance c. Let e be the number of input edges. We assume all distances are non-negative and that every source node is contained in at least one edge so that the number of nodes is at most twice the number of edges. The rule set derives assertions of the form $\mathsf{dist}(v, d)$ stating that the shortest path from the given source node to v is no longer than d. Note that rule (D2) ensures that in any state with priority 2 or greater we have at most one bound associated with each node. It is easy to see that when the computation terminates each bound is equal to the actual minimal distance — the procedure is correct. Furthermore, for any state transition cause by an instance of (D3) on node v and distance d we have that all future state transitions involve distances at least as large as d and so d is the final distance, i.e., the true shortest distance to d. This implies that all distance assertions are either of the form $\mathsf{dist}(v, 0)$, where v is the source node, or of the form $\mathsf{dist}(u, d+c)$ such that some edge ending in u with cost c starts a node with minimal distance d. This implies that there are at most $O(e)$ distance assertions ever asserted and that the final state (including all deleted assertions) is of size

$$\texttt{(B1,1)}\ \frac{E(v,u)}{E(u,v)} \qquad \texttt{(B2,1)}\ \frac{E(v,u)}{\mathsf{unlabeled}(u)} \qquad \texttt{(B3,1)}\ \frac{\mathsf{labeled}(u,k)}{\mathrm{del}(\mathsf{unlabeled}(u))}$$

$$\texttt{(B4,1)}\ \frac{\begin{array}{c}\mathsf{labeled}(u,A)\\ E(u,v)\end{array}}{\mathsf{labeled}(v,B)} \qquad \texttt{(B5,1)}\ \frac{\begin{array}{c}\mathsf{labeled}(u,B)\\ E(u,v)\end{array}}{\mathsf{labeled}(v,A)} \qquad \texttt{(B6,2)}\ \frac{\mathsf{unlabeled}(u)}{\mathsf{labeled}(u,A)}$$

Fig. 1. Checking for Bipartiteness.

$O(e)$. Now consider the abstract running time of this procedure. When rule (D3) derives a new distance, rule (D2) immediately deletes the larger distance. In a strong prefix firing all antecedents must be simultaneously true at the same state. This implies there is one strong prefix firing of (D2) for each state transition induced by rule (D3). Each state transitions caused by rule (D3) corresponds to a strong prefix firing of (D3) so the number of such transitions can be no larger than the number of strong prefix firings of (D3). So the number of strong prefix firings of (D2) is bounded by the number of strong prefix firings of (D3). To bound the number of strong prefix firings of (D3) we show that in any strong prefix firings of (D3) we must have that d is the unique shortest distance to the node v. To see this consider any strong prefix firing of (D3) involving v and d. By definition there must be a state S with $\pi(S) \geq d+2$ such that $\mathsf{dist}(v,d)$ holds in S. But for this rule set, if $\pi(S) \geq d+2$ then the next state transition must use an instance of (D3) involving a distance d or larger. This implies that all bounds derived after this point will also be of distance d or larger. So d must be the final distance bound for v, i.e., the true shortest distance to v. So in strong prefix firings of (D3) each node v is always associated with the same distance d. This implies that the number of strong prefix firings of (D3) is $O(e)$. So we have that $|S_0|$ is $O(e)$, P_f is $O(e)$, P_v is $O(e)$ and A_v (the number of antecedent instances of variable priority rules) is no larger than $|S_t|$ which is $O(e)$. So the abstract complexity is $O(e \log e)$.

Figure 3 shows a logic programming implementation of union-find. The rules maintain the invariant that the find relation is functional — for any x there is at most one y such that $\mathsf{find}(x,y)$ is visible. Furthermore, any two nodes x and y have the same find value if and only if they are equivalent in the smallest equivalence relation defined by the union operations. Figure 3 is essentially the implementation given in [2] and we do not give a detailed analysis here. This implementation uses greedy path compression. The total abstract running time

$$\texttt{(D1,1)}\ \frac{\mathsf{source}(v)}{\mathsf{dist}(v,0)} \qquad \texttt{(D2,1)}\ \frac{\begin{array}{l}\mathsf{dist}(v,d)\\ \mathsf{dist}(v,d')\\ d' < d\end{array}}{\mathrm{del}(\mathsf{dist}(v,d))} \qquad \texttt{(D3,}d\texttt{+2)}\ \frac{\begin{array}{l}\mathsf{dist}(v,d)\\ E(v,c,u)\end{array}}{\mathsf{dist}(u,d+c)}$$

Fig. 2. Dijkstra Shortest Path.

of all the union-find rules in figure 3 is $O(n \log n)$ where n is the total number of union operations in the computation, i.e., the total number of union assertions, including deleted assertions, in the final state. Furthermore, for any fixed node x, the number of nodes y such that $\mathsf{find}(x,y)$ is ever asserted is at most $\log n$ (every time the find of a node changes the size of that node's equivalence class at least doubles). It is possible to implement a lazy path compression version of union-find that has inverse-Ackermann running time but the implementation is somewhat more involved and the interface to the rules is more complex[1]

Figure 4 gives an algorithm for computing a minimum spanning tree of a connected undirected graph. These rules take a set of input arcs of the form $E(x,c,y)$ which states that there exists an undirected arc between x and y with cost c. The rules produce output arcs of the form $\mathsf{out}(x,c,y)$. The rules use the union-find module given in figure 3 to maintain an equivalence relation on nodes where x and y are equivalent if and only if there exists a path in the *output* arcs from x to y. The rules also maintain the invariant that the set of output edges form a subset of some minimum spanning tree. To see this consider an invocation of rule (ST2) with first antecedent $E(x,c,y)$. The edge in the first antecedent connects two unequal nodes and is minimum cost over all edges connecting nonequivalent nodes. By the induction hypothesis there exists a minimum spanning tree containing all existing output edges. Let S be such a minimum spanning tree. If S contains the edge $E(x,c,y)$ then this invocation of the rule preserves the invariant. If S does not contain $E(x,c,y)$ then consider the path in S from the equivalence class of x to the equivalence of y under the equivalence relation defined by the current output edges where each current equivalence class is viewed as a single node. Remove any edge in this path from S. The resulting graph has two components — one containing x and one containing y. Now add the edge $E(x,c,y)$. The result is a spanning tree of cost no larger than the original. This new spanning tree is also minimum. Hence there exists a minimum spanning tree containing $E(x,c,y)$ and the invariant is maintained. The output edges of the final state must then be a minimum spanning tree of the entire input graph.

[1] The rules in figure 3 take union assertions as input and produce find assertions as output. An inverse-Ackermann version requires on-demand find requests as additional input.

$$\texttt{(UF1,1)}\ \frac{\mathsf{union}(x,y)}{\begin{array}{l}\mathsf{nofind}(x)\\ \mathsf{nofind}(y)\end{array}} \qquad \texttt{(UF2,1)}\ \frac{\mathsf{find}(x,y)}{\mathrm{del}(\mathsf{nofind}(x))} \qquad \texttt{(UF3,1)}\ \frac{\begin{array}{l}\mathsf{find}(x,y)\\ \mathsf{find}(y,z)\end{array}}{\begin{array}{l}\mathsf{find}(x,z)\\ \mathrm{del}(\mathsf{find}(x,y))\end{array}}$$

$$\texttt{(UF4,1)}\ \frac{\begin{array}{l}\mathsf{union}(x,y)\\ \mathsf{find}(x,z)\\ \mathsf{find}(y,z)\end{array}}{\mathrm{del}(\mathsf{union}(x,y))} \qquad \texttt{(UF5,2)}\ \frac{\mathsf{nofind}(x)}{\begin{array}{l}\mathsf{find}(x,x)\\ \mathsf{size}(x,1)\end{array}} \qquad \texttt{(UF6,2)}\ \frac{\begin{array}{l}\mathsf{union}(x,y)\\ \mathsf{find}(x,x')\\ \mathsf{find}(y,y')\end{array}}{\mathsf{merge}(x',y')}$$

$$\texttt{(UF7,1)}\ \frac{\begin{array}{l}\mathsf{merge}(x,y)\\ \mathsf{size}(x,s1)\\ \mathsf{size}(y,s2)\\ s1 < s2\end{array}}{\begin{array}{l}\mathrm{del}(\mathsf{merge}(x,y))\\ \mathsf{find}(x,y)\\ \mathrm{del}(\mathsf{size}(y,s2))\\ \mathsf{size}(y,s1+s2)\end{array}} \qquad \texttt{(UF8,1)}\ \frac{\begin{array}{l}\mathsf{merge}(x,y)\\ \mathsf{size}(x,s1)\\ \mathsf{size}(y,s2)\\ s2 \leq s1\end{array}}{\begin{array}{l}\mathrm{del}(\mathsf{merge}(x,y))\\ \mathsf{find}(y,x)\\ \mathrm{del}(\mathsf{size}(x,s1))\\ \mathsf{size}(x,s1+s2)\end{array}}$$

Fig. 3. Union-Find

Now consider the abstract running time of the algorithm shown in figure 4. Let n be the number of nodes and e be the number of edges in the input graph. The rules can generate at most $n-1$ union assertions before all nodes become equivalent. Since there are at most $n-1$ union assertions, the abstract running time of the union-find rules is $O(n \log n)$. In any state of priority 2 or greater there is at most one find value for each node. Furthermore, for a given x there can be at most $\log n$ values y such that $\mathsf{find}(x,y)$ is ever asserted. This implies that there can be at most $O(e \log n)$ strong prefix firings of rule (ST1). Since this rule has a fixed priority, its abstract running time is $O(e \log n)$. Finally, there are at most e prefix firings and at most e antecedent instances of rule (ST2) and at most e different priorities associated with this rule. Hence the abstract running time of rule (ST2) is no larger than $O(e \log e)$. So we get a total abstract running time of $O(e \log e)$.

$$(\texttt{ST1,3})\ \frac{\begin{array}{l}E(x,c,y)\\ \mathsf{find}(x,z)\\ \mathsf{find}(y,z)\end{array}}{\mathrm{del}(E(x,c,y))} \qquad (\texttt{ST2,c+4})\ \frac{E(x,c,y)}{\begin{array}{l}\mathsf{union}(x,y)\\ \mathsf{out}(x,c,y)\end{array}}$$

Fig. 4. Minimum Spanning Tree

4 Proof of Theorem 1

We first use a series of run-time preserving rule transformations to put rules in a standard form. First we show that it suffices to consider rules with at most two antecedents and with no more than one assertion in the conclusion. We start by replacing the given priority assignment π by the assignment π' where $\pi'(r) = 2 * \pi(r) + 2$. In the resulting program every priority is even and no rule has priority 2. Now consider a rule r of the form $A_1 \wedge \cdots \wedge A_n \Rightarrow \{\Phi_1, \ldots, \Phi_m\}$. We replace r by the following where P_1, P_2, ..., P_n, and Q are fresh predicate symbols and x_1, ..., x_{k_i} are the variables occurring in the first i antecedents. The predicate P_i represents the relation defined by the first i antecedents, and $\mathrm{del}(P_i(x_1,\ \ldots,\ x_{k_i}))$ represents the statement that some antecedent has been deleted.

$$\begin{aligned}
A_1 &\Rightarrow P_1(x_1, \ldots, x_{k_1})\\
\mathrm{del}(A_1) &\Rightarrow \mathrm{del}(P_1(x_1, \ldots, x_{k_1}))\\
P_1(x_1, \ldots, x_{k_1}) \wedge A_2 &\Rightarrow P_2(x_1, \ldots, x_{k_2})\\
P_2(x_1, \ldots, x_{k_2}) \wedge \mathrm{del}(P_1(x_1, \ldots, x_{k_1})) &\Rightarrow \mathrm{del}(P_2(x_1, \ldots, x_{k_2}))\\
P_2(x_1, \ldots, x_{k_2}) \wedge \mathrm{del}(A_2) &\Rightarrow \mathrm{del}(P_2(x_1, \ldots, x_{k_2}))\\
&\ \vdots\\
P_{n-1}(x_1, \ldots, x_{k_1}) \wedge A_n &\Rightarrow P_n(x_1, \ldots, x_{k_2})\\
P_n(x_1, \ldots, x_{k_2}) \wedge \mathrm{del}(P_{n-1}(x_1, \ldots, x_{k_1})) &\Rightarrow \mathrm{del}(P_n(x_1, \ldots, x_{k_2}))\\
P_n(x_1, \ldots, x_{k_2}) \wedge \mathrm{del}(A_n) &\Rightarrow \mathrm{del}(P_n(x_1, \ldots, x_{k_2}))\\
P_n(x_1, \ldots, x_{k_n}) &\Rightarrow Q(x_1,\ \ldots, x_{k_n})\\
Q(x_1, \ldots, x_{k_n}) &\Rightarrow \Phi_1\\
&\ \vdots\\
Q(x_1, \ldots, x_{k_n}) &\Rightarrow \Phi_m
\end{aligned}$$

If some antecedent of the original rule r is of the form $\mathrm{del}(B_j)$ then the above rules include rules with double deleted antecedents, i.e., rules with antecedents of the form $\mathrm{del}(\mathrm{del}(B_j))$. Rules with double deletion antecedents are simply thrown out — they are not included in the result of the transformation (deletion assertions can not be deleted). Rules containing deletions of arithmetic comparisons, e.g., $\mathrm{del}(i < j)$ are also ignored. All rules with conclusions of the form $\mathrm{del}(P_i(x_1, \ldots, x_{k_i}))$ or Φ_j have priority 1; rules with conclusions of the form $P_i(x_1, \ldots, x_{k_i})$ have priority $\pi(r) - 1$, and the rule $P_n(x_1,\ldots,x_{k_n}) \Rightarrow Q(x_1,\ldots,x_{k_n})$ has priority $\pi(r)$. In a computation under the transformed program, states of the original computation correspond even priority states of the transformed program. Odd priority states of the transformed program are used to compute the next even-priority state. We leave it to the reader to verify that the sequence of even priority states, restricted to the predicates of the original program, yields a valid computation of the original program.

We also have to argue that the abstract running time of the transformed program is no more than a constant factor more than the abstract running time of the original program. We do this by charging each prefix firing of the transformed program to some prefix firing of the original program. Since the deletion rules run at the highest possible priority, if $P_i(x_1, \ldots, x_{k_i})$ ever holds in any state then there exists a prefix firing in the even-state computation corresponding to this assertion. This implies that a firing of a rule whose conclusion is of the form $\mathrm{del}(P_i(x_1, \ldots, x_{k_i}))$ can be charged to the prefix of r corresponding to the assertion being deleted in the conclusion. We also have that if $P_i(x_1, \ldots, x_{k_i})$ holds in a state with priority $\pi(r) - 1$ then the corresponding prefix firing of r must hold in that state. This implies that each firing of a rule whose conclusion is of the form $\mathrm{del}(P_i(x_1, \ldots, x_{k_i}))$ corresponds to a prefix firing of the original rule. Each firing of $P_n(x_1,\ldots,x_{k_n}) \Rightarrow Q(x_1, \ldots, x_{k_n})$ or a rule with a conclusion of the form Φ_j corresponds to a full firing of the original rule. A similar argument can be used to show that the number antecedent instances of variable priority rules of the new program is no more than a constant times the number of antecedent instances of variable priority rules of the old program.

We have now shown that we can assume without loss of generality that each rule contains at most two antecedents and only a single conclusion. We now simplify the rules further. Starting with a set of rules with at most two antecedents and only a single conclusion replace the priority assignment π by π' where $\pi'(r) = \pi(r) + 1$. Consider a rule r with two antecedents $A_1 \wedge A_2 \Rightarrow B$. The definition of an inference rule requires that any variable in a comparison antecedent occurs in some earlier antecedent. This implies that the first antecedent is not a comparison. We first consider the case where the second antecedent is also not a comparison — the case where the second antecedent is a comparison is discussed below. If the second antecedent is not a comparison we replace r by the following set of rules where $x_1, \ldots, x_n$ are all variables occurring in A_1 but not A_2, $y_1, \ldots, y_m$ are all variables that occur in both A_1 and A_2, and $z_1, \ldots, z_k$ are all variables that occur in A_2 but not A_1. The predicates P, and Q, and the function symbols f, g, and h are all fresh.

$$\begin{aligned}
A_1 &\Rightarrow P(f(x_1, \ldots, x_n), g(y_1, \ldots, y_m)) \\
\mathrm{del}(A_1) &\Rightarrow \mathrm{del}(P(f(x_1, \ldots, x_n), g(y_1, \ldots, y_m))) \\
A_2 &\Rightarrow Q(g(y_1, \ldots, y_m), h(z_1, \ldots, z_k)) \\
\mathrm{del}(A_2) &\Rightarrow \mathrm{del}(Q(g(y_1, \ldots, y_m), h(z_1, \ldots, z_k))) \\
P(f(x_1, \ldots, x_n), g(y_1, \ldots, y_m)) \wedge Q(g(y_1, \ldots, y_m), h(z_1, \ldots, z_k)) &\Rightarrow B
\end{aligned}$$

The last rule is given the same priority as the original and the other rules are given priority one. The states with priority greater than one correspond to the states of the original rule set. We can charge the firings of the rules with a single antecedent of the form A_1, A_2, $\mathrm{del}(A_1)$, or $\mathrm{del}(A_2)$ to the rules asserting A_1 or A_2.

We can now assume without loss of generality that every rule is either of the form $A \Rightarrow B$ where A is not a comparison, or of the form $A_1 \wedge A_2 \Rightarrow B$ where A_1 is not a comparison, and either A_2 is a comparison with all variables of A_2 occurring in A_1, or else the rule is of the form $P(v, v') \wedge Q(v', v'') \Rightarrow B$ where the three terms v, v', and v'' do not share any variables. These rules will be called *unary rules*, *comparison rules*, and *binary rules* respectively.

We now define a *firing tuple* to be a tuple of the form $\langle r,\ A_1\rangle$ or $\langle r,\ A_1,\ A_2\rangle$ where r is a rule and A_1 is a ground assertion matching the first antecedent of r and A_2 is a ground assertion matching the second antecedent of r. To implement the saturation algorithm we maintain a global priority queue $\mathcal{Q}$ of not-yet-processed firing tuples and a set $\mathcal{R}$ of already-processed firing tuples. The priority of a tuple involving r and A_1 is defined to be $\pi(r,\ \sigma)$ where σ is the substitution resulting from matching the first antecedent of r with A. The algorithm iteratively removes and processes tuples from $\mathcal{Q}$. In addition, for each binary rule r of the form $P(v, v') \wedge Q(v', v'') \Rightarrow B$, and for certain ground terms t we maintain a data structure $\mathcal{W}(r,\ t)$ representing potential full firings of r of the form $P(s, t)$, $Q(t, u)$. More specifically, $\mathcal{W}(r,\ t)$ is an alternating sequence of P-blocks and Q-atoms where a P-block is a possibly empty set of ground P-atoms of the form $P(s, t)$ matching $P(v, v')$ and a Q-atom is an ground assertion of the form $Q(t, u)$ matching $Q(v', v'')$ and any given assertion appears in $\mathcal{W}(r,\ t)$ in at most one place. The sequence $\mathcal{W}(r,\ t)$ starts and ends with (possibly empty) P-blocks. Atoms in the last P-block are called *completed*. The priority of an element $P(s, t)$ of a P-block of $\mathcal{W}(r,\ t)$ is $\pi(r,\ \sigma)$ where σ is the corresponding substitution. Each nonempty P-block is associated with a unique element of highest priority with ties brocken arbitrarily. We will refer to this selected element simply as "the" highest priority element of the P-block. The saturation procedure maintains the following $\mathcal{Q}$ invariant which determines the set of tuples in $\mathcal{Q}$ as a function of the current state S, the set $\mathcal{R}$ of processed firings, and the state of the data structures of the form $\mathcal{W}(r,\ t)$.

> **$\mathcal{Q}$-Invariant:** The queue $\mathcal{Q}$ contains a pair of the form $\langle\ r,\ A\rangle$ if and only if A is visible in the current state; r is a rule in the program whose first antecedent matches A; and the pair $\langle\ r,\ A\rangle$ is not contained on the list $\mathcal{R}$. An triple of the form $\langle\ r,\ P(s, t),\ Q(t, u)\rangle$ is on the queue $\mathcal{Q}$ if and only if $P(s, t)$ and $Q(t, u)$ are both contained in the data structure $\mathcal{W}(r,\ t)$ and $P(s, t)$ is a highest priority element of the P-block immediately preceding the Q-assertion $Q(t, u)$.

Procedure $\mathsf{Assert}(A)$
If A is not visible in S do the following
1. Add A to S.

2. If A is of the form del(B) then remove B and tuples involving B from $\mathcal{Q}$ and all data structures of the form $\mathcal{W}(r,\ t)$. For each Q-assertion removed from a data structure of the form $\mathcal{W}(r,\ t)$ merge the two adjacent P-blocks updating $\mathcal{Q}$ as necessary to maintain the $\mathcal{Q}$-invariant.
3. For each rule r such that A matches the first antecedent of r add the pair $\langle r,\ A\rangle$ to $\mathcal{Q}$.
4. For each binary rule r such that A matches the second antecedent of r add A as a new Q-atom at the end of $\mathcal{W}(r,\ t)$, where t is the first argument of A, followed by a new empty P-block (initialize $\mathcal{W}(r,\ t)$ with an empty P-block if necessary). If the P-block preceding Q is non-empty add the triple $\langle r,\ B,\ A\rangle$ to $\mathcal{Q}$ where B is the highest priority P-assertion in the P-block preceding Q.

Saturation Procedure Let D be a given initial database.

1. Initialize $\mathcal{Q}$ and the state S to be empty.
2. Assert each element of D.
3. While $\mathcal{Q}$ is not empty do the following:
   (i) Remove the highest priority tuple z on $\mathcal{Q}$ and add it to $\mathcal{R}$.
   (ii) If z is $\langle r,\ A\rangle$ for unary r assert the corresponding conclusion of r.
   (iii) If z is $\langle r,\ A\rangle$ for a comparison rule r assert the corresponding conclusion of r provided that the comparison holds.
   (iv) If z is $\langle r,\ P(s,t)\rangle$ where r is a binary rule then add $P(s,t)$ to the first P-block of $\mathcal{W}(r,\ t)$ (initialize $\mathcal{W}(r,\ t)$ if necessary). Update $\mathcal{Q}$ so as to maintain the $\mathcal{Q}$-invariant over this change to $\mathcal{W}(r,\ t)$.
   (v) If z is $\langle r,\ P(s,t),\ Q(t,\ u)\rangle$, assert the corresponding conclusion of r and move $P(s,t)$ from the P-block preceding $Q(t,u)$ to the one following and update $\mathcal{Q}$ so as to maintain the $\mathcal{Q}$-invariant.

It is easy to check that the saturation procedure is sound — each state transition corresponds to the invocation of a highest priority rule instance. It is also easy to check that the procedure does not terminate until a saturated state has been reached. We now consider the running time of the saturation procedure. We assume that all expressions are "interned", also known as "hash consed", so that the same expression is always represented by the same pointer to memory. This allows equality testing to be done in unit time. Assuming that hash table operations take unit time, matching and instantiating a given pattern can also be done in unit time. Computing the priority of a pair $\langle r,\ \sigma\rangle$ or $\langle r,\ A\rangle$ can also be done in unit time. Let k be the lowest priority (largest number) of any fixed priority rule in the given program. We implement $\mathcal{Q}$ as a pair of a priority queue for items of priority k or less and a Fibonacci heap for higher priorities. For priorities of k or less the operations of insertion, deletion, and merging of queues all take unit time. For larger priorities the Fibonacci heaps support $O(1)$ amortized insertion and queue merger and $O(\log N)$ amortized removal and finding the highest priority element.[2] The P-blocks are implemented as doubly

[2] The amortized time for removal and find-min operations in fibonacci heaps is usually given as $O(\log n)$ where n is the number of elements on the queue. By using at

linked lists for fixed priority rules and as Fibonacci heaps for variable priority rules.

First we show that the total amount of time spent inside the assert procedure is $O(|D| + P_f + (P_v + A_v) \log N)$. Each call to the assert procedure either corresponds to an element of D or to a unique strong prefix firing corresponding to the value of z in step (i) of the saturation procedure. Since we are assuming that expressions are interned we can determine whether an expression is already a member of S by checking for an appropriate field in the data structure for that assertion. So asserting an expression that is already visible takes $O(1)$ time. For a new assertion we consider the cost of each step in the assert procedure. Step 1 involves simply setting the value of an appropriate field in the data structure for the assertion and takes $O(1)$ time. To analyze the cost of removing B in step 2 we note that for any given assertion B and rule r there is at most one triple of the form $\langle r,\ A,\ B\rangle$ and at most one triple of the form $\langle r,\ B,\ A\rangle$ in $\mathcal{Q}$. This implies that the total number of removals in step 2 is $O(1)$. For variable priority rules, a single removal costs $O(\log N)$ (amortized) including the time spent merging P-blocks and updating $\mathcal{Q}$ to maintain the $\mathcal{Q}$-invariant. However, a single assertion can only be removed once and hence the total time spent removing instances of antecedents of variable priority rules is $O(A_v \log N)$. So the total time in step 2 is $O(|D| + P_f + P_v + A_v \log N)$. We now consider step 3 of the assert procedure. Since Fibonacci heaps support $O(1)$ time (amortized) insertions, the total time in step 3 is $O(D + P_f + P_v)$. A similar observation applies to step 4.

Finally we consider the time spent executing the saturation procedure not counting time spent inside calls to assert. The time for steps 1 and 2 is $O(D)$. Each iteration of step 3 corresponds to a strong prefix firing of some rule. Step (i) of 3 takes $O(1)$ time if the rule involved is constant priority and $O(\log N)$ time if the rule involved is variable priority. So the total time taken by the removals in substep (i) is $O(P_f + P_v \log N)$. Substeps (ii) and (iii) are $O(1)$ time per iteration. We now consider substep (iv). Insertion into a P-block takes $O(1)$ time (amortized). This insertion may involve a removal and an insertion into $\mathcal{Q}$ in order to maintain the $\mathcal{Q}$-invariant. The insertion takes $O(1)$ but in the case where the rule is variable-priority the removal can take $O(\log N)$ time. Fortunately, in this case the prefix firing corresponding to the value of z is a variable-priority firing. So the time taken in step (iv) is $O(P_f + P_v \log N)$. A similar analysis holds for substep (v). We now have the total execution time is $O(|D| + P_f + (P_v + A_v) \log N)$.

5 Conclusion

We have given a broad scope algorithmic logic programming model of computation. The increased scope of the model over those given in [5] and [2] has come at the cost of some increased complexity in both the semantics of the language

most one element for each priority, where that element is a doubly linked list of "subelements", we can implment removal and find-min in $O(\log N)$ amortized time where N is the number of distinct priorities.

and in the associated notion of abstract running time for logic programs. We are hopeful, however, that this language has sufficient scope to cover the majority of algorithms found in standard texts. It would indeed be interesting to write an algorithms text based on an algorithmic logic programming model of computation.

References

1. Weidong Chen and David S. Warren. Tabled evaluation with delaying for general logic programs. *Journal of the ACM*, 43(1):20–74, 1996.
2. H. Ganzinger and D. McAllester. A new meta-complexity theorem for bottom-up logic programs. In *Proc. International Joint Conference on Automated Reasoning*, volume 2083 of *Lecture Notes in Computer Science*, pages 514–528. Springer-Verlag, 2001.
3. Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In Robert A. Kowalski and Kenneth Bowen, editors, *Proceedings of the Fifth International Conference on Logic Programming*, pages 1070–1080, Cambridge, Massachusetts, 1988. The MIT Press.
4. Kolaitis. The expressive power of stratified logic programs. *INFCTRL: Information and Computation (formerly Information and Control)*, 90, 1991.
5. David McAllester. The complexity analysis of static analyses. In *Symposium on Static Analysis*. Springer Verlag, 1999.
6. Jeff Naughton and Raghu Ramakrishnan. Bottom-up evaluation of logic programs. In Jean-Louis Lassez and Gordon Plotkin, editors, *Computational Logic*. MIT Press, 1991.
7. Jeff Polakow. Ordered linear logic and applications, 2001. Ph.D. Dissertation, Carnegie Mellon University.
8. T. Przymusinski. the declarative semantics of stratified deductive databases and logic programs, 1988.
9. K. Sagonas, T. Swift, and D. S. Warren. Xsb as an efficient deductive database engine. In *Proc. of the 1994 ACM SIGMOD Int. Conf. on Management of Data (SIGMOD'94)*, 1994.
10. H. Tamaki and T. Sato. Old resolution with tabulation. In *Proceedings of the Third International Conference on Logic Programming*, 1986.
11. J. Ullman. Bottom-up beats top-down for datalog. In *Proceedings of the Eigth ACM SIGACT-SIGMOD-SIGART Symposium on the Principles of Database Systems*, pages 140–149, March 1989.
12. Jeffrey Ullman and Raghu Ramakrishnan. A survey of research in deductive database systems. *J. Logic Programming*, pages 125–149, May 1995.
13. M. Vardi. Complexity of relational query languages. In *14th Symposium on Theory of Computation*, pages 137–146, 1982.
14. Yan Zhang and Norman Y. Foo. Towards generalized rule-based updates. In *IJCAI-97*, 1997.