Processing Boolean Equalities and Inequalities

Jayadev Misra*

October 24, 2011

1 Introduction

We are given a set of boolean variables, and a set of equalities of the form $x \equiv y$ and inequalities of the form $x \not\equiv y$, over these variables. We call each equality and inequality a *fact*. A query is of the form $x \equiv y?$. Based on the given facts, an outcome of a query is "*true*" (that is, $x \equiv y$), "*false*" (that is, $x \not\equiv y$), and "$\perp$" (that is, no relationship between x and y can be deduced from the given facts). An inconsistency arises if both $x \equiv y$ and $x \not\equiv y$ can be deduced from the given facts for some pair of variables x and y; inconsistency should be reported.

We propose an offline and an online algorithm to process facts and answer queries. In the offline version, queries are posed only after all facts have been presented. In the online version, the facts and queries are intermingled; answer to each query is based on the facts presented so far. The offline version processes each fact in constant time; then, after all the facts have been processed, it builds a data structure in time linear in the number of variables, and then answers each query in constant time. The online version is based on a modification of the well-known union-find algorithm (see chapter 21 on "Data Structures and Disjoint Sets" in [1]), and has the same time bounds as that algorithm.

The complexity of this problem is no less than processing equivalence relations over arbitrary domains, not just booleans[1]. To see this, consider the special case where all facts are equalities. We claim that this problem is equivalent to the equivalence problem in which the variables are not restricted to be boolean. For any instance of the general equivalence problem, construct the corresponding boolean equivalence problem (where all facts are now equalities). A *derivation* of $x_0 \equiv x_n$ in either domain is a sequence $x_0 \equiv x_1$, $x_1 \equiv x_2$, $\cdots$, $x_{n-1} \equiv x_n$, where each $x_i \equiv x_{i+1}$ is a fact or $x_{i+1} \equiv x_i$ is a fact. Answer to a query is based on a derivation. It is easy to see that a derivation exists in one domain iff it exists in the other domain. Therefore, restricting values to be boolean is no less general. The problem we study adds the complication of inequalities.

*This work is partially supported by National Science Foundation grant CCF-0811536.

[1]Observation due to Greg Plaxton.

The problem is easily solved by coding the facts and queries in a reasoning language, such as CCalc[4], and then applying a SAT solver. These are powerful tools that can process far more general facts and queries. Frühwirth [3] contains important generalizations of the classical union-find algorithm that are applicable to arbitrary relations; his work can be applied to the problem considered here. Ours is a much simpler solution for this specific problem.

2 Basic Results

Both offline and online algorithms exploit the following results about boolean equalities and inequalities. Define $x \sim y$ to mean that either $x \equiv y$ or $x \not\equiv y$.

Proposition 1 Relation $\sim$ is an equivalence relation.

Proof: (Reflexivity) For any x, $x \sim x$ follows from $x \equiv x$. (Symmetry) Follows from the symmetry of both $\equiv$ and $\not\equiv$. (Transitivity) Given $x \sim y$ and $y \sim z$, if both facts are equalities or both inequalities, then $x \equiv z$; hence, $x \sim z$. If one of the facts is an equality and the other inequality, then $x \not\equiv z$; hence, $x \sim z$. $\Box$

We exploit Proposition 1 to encode the facts in a graph such that $x \sim y$ iff x and y belong to the same connected component. And, we label each edge (x, y) with the value of $x \equiv y$. Proposition 3, below, shows that the value of $(x \equiv y)$, for any two nodes x and y in a component of the graph, can be computed by applying $\equiv$ to the edge labels along a path between these nodes.

Proposition 2 The $\equiv$ relation has the following properties.

$$\begin{array}{lrcl}
\text{Commutativity:} & x \equiv y & = & y \equiv x \\
\text{Associativity:} & x \equiv (y \equiv z) & = & (x \equiv y) \equiv z \\
\text{Zero:} & (x \equiv x) \equiv z & = & z
\end{array}$$

Proof: These are standard properties of equivalence relations; see Dijkstra and Scholten [2]. $\Box$

Proposition 3 Consider a graph in which the label of every edge (p, q) is the value of $p \equiv q$. For any two nodes x and y in the graph, the value of $x \equiv y$ can be computed by applying $\equiv$ to the edge labels along any path between x and y.

Proof: Let $x = x_0,\ x_1,\ \cdots,\ x_n = y$ be a path connecting x and y.

$$\begin{array}{ll}
 & (x_0 \equiv x_1) \equiv (x_1 \equiv x_2) \cdots \equiv (x_{n-1} \equiv x_n) \\
= & \{\text{Rewrite using commutativity and associativity of } \equiv\} \\
 & (x_0 \equiv x_n) \equiv (x_1 \equiv x_1) \cdots \equiv (x_{n-1} \equiv x_{n-1}) \\
= & \{\text{Drop every } (x_i \equiv x_i) \text{ term}\} \\
 & (x_0 \equiv x_n) \\
= & \{x = x_0 \text{ and } x_n = y\} \\
 & (x \equiv y)
\end{array}$$

$\Box$

3 Offline Algorithm

Construct an undirected graph where each node corresponds to a variable. Given a fact $x \equiv y$ or $x \not\equiv y$, introduce edge (x, y) and label it with the value of $x \equiv y$, i.e., set $(x, y).l := (x \equiv y)$, where the label of (x, y) is $(x, y).l$. We may assume that no fact is of the form $x \equiv x$ (redundant) or $x \not\equiv x$ (inconsistent). Therefore, there is no self-loop in the graph.

Nodes x and y are connected iff $x \sim y$, that is, the connected components of the graph correspond to the equivalence classes under $\sim$. After all the facts have been presented, we process the graph to build a data structure that allows answering each query in constant time.

The given facts are consistent iff there is an assignment of boolean values to the variables, i.e., nodes of the graph, satisfying the facts. It is sufficient to check each component of the graph independently for consistency. Pick an arbitrary node r in a component and assign it an arbitrary boolean value $r.l$. Then for each edge (x, y), assign values to x and y such that $x.l \equiv ((x, y).l \equiv y.l)$. If the variables can be assigned unique values so that this assertion holds for each edge then the facts are consistent, by definition. If some node x gets two different assigned values, *true* and *false*, then the facts are inconsistent. This is because, considering the two paths from r to x that caused these assignments, we derive $r.l \equiv \textit{true}$ and $r.l \equiv \textit{false}$, leading to a contradiction.

For node x, let $x.c$ be the identity of the component to which it belongs. The answer to the query $(x \equiv y)$?, given that the facts are consistent, is

$$\begin{cases} \perp & \text{if } x.c \neq y.c \\ x.l \equiv y.l & \text{otherwise} \end{cases}$$

Efficiency Each fact creates a single labeled edge of the graph, so each fact is processed in constant time. Identifying the connected components and labelling the nodes takes linear time, which includes the consistency check. If the facts are consistent, each query is then answered in constant time.

4 Online Algorithm

The online problem, as described in section 1, intermingles facts and queries. We will use a variation of the *union-find* algorithm for this case. Below, we give a brief description of this algorithm, which is necessary for understanding its extension to the boolean domain with equalities and inequalities.

4.1 union-find algorithm

The algorithm represents each connected component of the graph by a tree over the nodes in the component. Each node x has a field $x.\textit{parent}$ for the identity of its parent (the parent of a root is $\textit{nil}$). The algorithm supports two fundamental operations: (1) $\textit{union}(r, s)$, where r and s are the roots of two different trees, merges these two trees, and (2) *find*(x) returns the root of the tree to which x belongs.

Implementing find with path compression To implement *find*(x), start at x and follow the sequence of parents to the root r of the tree. The following heuristic speeds up the algorithm. The path from x to r is followed again making each ancestor y of x, $y \neq r$, a child of r (a node is its own ancestor). Path compression doubles the cost of operation *find*, but it reduces the processing time for subsequent facts and queries.

Implementing union by rank To implement *union*(r, s), $r.parent$ is set to s, or vice versa; the choice is immaterial for the correctness of the algorithm. Choosing r to be the child if it is the root of a smaller tree (i.e., having fewer nodes) improves the performance of the algorithm.

The two heuristics can be applied independently. Applying both heuristics results in a running time that is almost linear in the number of facts and queries.

4.2 Extending the union-find algorithm

As before, we attach a boolean label $(x, y).l$ to every tree edge (x, y); the label has the value $x \equiv y$.

Implementing find The operation *find*(x) returns a pair (r, b) where r is the root of the tree to which x belongs and $b \equiv (x \equiv r)$. We describe how b is computed and path compression implemented.

Let $x_0, x_1, \cdots, x_n$ be the path from x to r where $x_0 = x$ and $x_n = r$. Now b equals $(x \equiv r)$, which, using Proposition 3, can be computed from the edge labels along the path from x to r. To implement path compression, each x_i is made a child of r. The edge label of (x_i, r) is $x_i \equiv r$, which is computed iteratively as follows. By the time r has been computed, $x_0 \equiv r$ is known. For $i > 0$, $(x_i \equiv r) = ((x_i \equiv x_{i-1}) \equiv (x_{i-1} \equiv r))$. The first term is the label of the edge from x_{i-1} to its previous parent x_i; the second term has been computed earlier.

Implementing union The union *operation* has an additional boolean argument; the effect of executing *union*(r, s, b), where r and s are roots of different trees, is to make r the parent of s, or vice versa, and label the edge that is introduced with b. The rank heuristic is used, as before, to choose which of r and s becomes the root of the merged tree.

Processing a fact, $x \equiv y$ or $x \not\equiv y$

Let $(xr, xb) :=$ *find*(x) and $(yr, yb) :=$ *find*(y). If $xr = yr$ then the nodes are in the same tree; i.e., the relationship between x and y is already known. If $xb \equiv yb$ equals $x \equiv y$ then the new fact is consistent and redundant. If $xb \equiv yb$ does not equal $x \equiv y$ then the new fact is inconsistent with the known facts. Otherwise,

$xr \neq yr$; we need to merge the trees with root xr and yr and assign the label $xr \equiv yr$ to the newly introduced edge. Apply $union(xr, yr, xb \equiv (x \equiv y) \equiv yb)$. To see the validity of the last parameter in the call to union, note that

$$\begin{array}{ll} & xb \equiv (x \equiv y) \equiv yb \\ = & \{\text{Using commutativity and associativity of } \equiv\} \\ & (xb \equiv x) \equiv (y \equiv yb) \\ = & \{xr = (xb \equiv x) \text{ and } yr = (y \equiv yb)\} \\ & xr \equiv yr \end{array}$$

Processing a query, $x \equiv y$?
Let $(xr, xb) := \mathit{find}(x)$ and $(yr, yb) := \mathit{find}(y)$. If $xr \neq yr$ then the nodes are in different trees; return $\perp$. If $xr = yr$, return $xb \equiv\ yb$. This is because,

$$\begin{array}{ll} & xb \equiv yb \\ = & \{xb = x \equiv xr,\ yb = y \equiv yr\} \\ & (x \equiv xr) \equiv (y \equiv yr) \\ = & \{\text{Using commutativity and associativity of } \equiv\} \\ & (x \equiv y) \equiv (xr \equiv yr) \\ = & \{xr = yr\} \\ & x \equiv y \end{array}$$

Acknowledgment The power of the equivalence relation was shown to me by Edsger W. Dijkstra. The arguments in this note would have been considerably more complicated had I not freely used the properties of the $\equiv$ operator.

I am thankful to Vladimir Lifschitz and Jonathan Campbell for acquainting me with their reasoning language Ccalc. Rutger M. Dijkstra and David Gries have provided extensive editorial comments on an earlier draft. Particular thanks to Greg Plaxton for many insightful comments.

References

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms.* McGraw Hill and MIT press, second edition, 2001.

[2] E.W. Dijkstra and C.S. Scholten. *Predicate Calculus and Program Semantics.* Texts and Monographs in Computer Science. Springer-Verlag, 1989.

[3] Thom Frühwirth. Quasi-linear-time algorithms by generalisation of union-find in CHR. In *Recent Advances in Constraints — CSCLP '07: 12th ERCIM Intl. Workshop on Constraint Solving and Constraint Logic Programming, Revised Selected Papers*, pages 91–118, November 2008.

[4] The home page for The Causal Calculator, CCalc. `http://www.cs.utexas.edu/users/tag/cc/`, 1997.