Information Processing Letters 21 (1985) 269–272
North-Holland

18 November 1985

A FAST ALGORITHM FOR RENAMING A SET OF CLAUSES AS A HORN SET

Heikki MANNILA

Department of Computer Science, University of Helsinki, Tukholmankatu 2, SF-00250 Helsinki 25, Finland

Kurt MEHLHORN

Fachbereich 10 — Informatik, Universität des Saarlandes, D-6600 Saarbrücken, Fed. Rep. Germany

Communicated by J. Nievergelt
Received 30 November 1984
Revised February 1985

A set of clauses is a Horn set if each clause contains at most one positive literal. Lewis (1978) has given a polynomial-time algorithm for testing whether a set of clauses can be renamed as a Horn set. His algorithm uses in the worst case $O(c \cdot v^2)$ time, where c is the number of clauses and v the number of variables. We give an algorithm working in $O(c \cdot v \cdot (\log v)^2)$ time. The algorithm is based on an efficient depth-first search on a dense graph with a short description.

Keywords: Horn clause, renaming, depth-first search

1. Introduction

A clause (a set of literals) is a *Horn clause* if it contains at most one positive literal. Sets of Horn clauses (also called *Horn sets*) have a polynomial satisfiability problem. They naturally occur in, e.g., logic programming [4] and database theory [3].

The class of Horn sets can be significantly extended and still retain the polynomiality of the satisfiability problem. Let S be a set of clauses and A a set of variables. The *A-renaming of* S, $r_A(S)$, is the result of replacing in S each literal whose variable is in A by its complement. The set S is *renamable-Horn* if $r_A(S)$ is Horn for some A.

Testing whether a set S is renamable-Horn is by no means trivial. Lewis [5] gave a polynomial-time algorithm for this problem. His algorithm also produces a renaming if one exists, and thus it shows that renamable-Horn formulas have a polynomial satisfiability problem.

Lewis' algorithm is based on the following elegant result: if

$$S = \{C_1, \ldots, C_m\} \quad \text{and} \quad C_i = \{L_{i1}, \ldots, L_{ik(i)}\},$$

then S is renamable-Horn if and only if the set

$$S^* = \bigcup_{i=1}^{m} \{\{L_{ij}, L_{ik}\} \mid 1 \leqslant j < k \leqslant k(i)\}$$

is satisfiable.

Since S^* contains at most two literals per clause, its satisfiability can be tested in linear time [2]. This is done by constructing a directed graph $G(S^*)$, whose vertex set is

$$\{x, \bar{x} \mid x \text{ is a variable of } S\},$$

and whose edge set is

$$\{(\bar{L}, M), (\bar{M}, L) \mid \{L, M\} \text{ is a clause of } S^*\}.$$

Now, S^* is satisfiable if and only if for no literal L both L and $\bar{L}$ are in the same strongly connected component of $G(S^*)$. The strongly connected components are found using depth-first search [1,6], which is linear in the size of the graph.

Thus, Lewis' result leads to a straightforward algorithm for testing renamability, with worst-case complexity $O(c \cdot v^2)$, where v is the number of variables and c the number of clauses. Note that the size of the input is O(c · v). The desired renaming is obtained from the truth assignment satisfying S*.

In this paper we describe an improved algorithm for testing whether a formula is renamable-Horn, with worst-case running time $O(c \cdot v \cdot (\log v)^2)$. The algorithm is based on Lewis' result, but it avoids the explicit construction of S* and G(S*). We still test satisfiability of S* using depth-first search on G(S*), but with the concise representation of G(S*) given by S itself. This makes it possible to avoid considering all edges of G(S*).

Section 2 of this paper discusses our refinement of depth-first search. It shows that for any graph one has to consider at most a linear number of edges to determine the strongly connected components. Section 3 shows how we can quickly find these edges.

2. Useful edges in depth-first search

We assume the reader to be familiar with depth-first search (see, e.g., [1] or [6]). It divides the edges of the graph into four disjoint classes: tree edges, forward edges, cross edges, and backward edges.

The search algorithm also gives a changing partition of vertices. At each point of the algorithm the cells of this partition contain the vertices which are known to belong to the same strongly connected component. In the beginning each cell contains just one vertex; after the algorithm terminates, the cells are exactly the strongly connected components. In usual depth-first search, the information about this partition is implicit; we will use an explicit representation.

The partition changes when two of its cells are merged. This happens when an edge is found which makes the cells equal. There are thus only a linear number of changes, since each change decreases the number of cells by one. Specifically, there are at most n − 1 edges which force us to merge two components, where n is the number of vertices.

Suppose depth-first search has proceeded to node v. Call an edge (v, w) *useful* if either it is a tree edge or following it forces us to merge two cells. Since there are at most n − 1 tree edges, there are at most 2n − 2 useful edges. The useful edges determine the same strongly connected components as the original edges. Therefore, one has to consider only useful edges in depth-first search. Our strategy is to use data structures which enable us to find these edges fastly.

Next we show how useful edges can be recognized on the basis of the partition. Assume we are currently inspecting node v. Denote by cell(u) the cell of the partition containing vertex u, for all u. Let (v, w) be an edge in the graph. We claim that it is a useful edge if and only if cell(v) ≠ cell(w) and the component including w is alive, i.e., it has not been output yet.

Suppose (v, w) is useful. If it is a tree edge, then cell(v) ≠ cell(w), since cell(w) = {w}, and it cannot be output yet. If (v, w) causes two components to be merged, the components must be disjoint at this stage, i.e., cell(v) ≠ cell(w). Since v belongs to the same strongly connected component as w, the strongly connected component containing w cannot be output yet.

For the converse, suppose (v, w) is not useful, i.e., it is a forward, backward, or cross edge which does not force us to merge cell(v) and cell(w). Assume first (v, w) is a forward edge. If cell(w) is alive, we must have cell(v) = cell(w), since the search from w has already been completed.

Assume then (v, w) is a backward edge. In this case, cell(w) is alive and there is a path from w to v. Since (v, w) does not merge cell(v) and cell(w), we must already have cell(v) = cell(w).

Finally, assume that (v, w) is a cross edge. If cell(w) is alive, then there is a path from w to v. Since (v, w) is not useful, we have already cell(v) = cell(w).

This characterization of useful edges shows that they can be recognized by using information about the cells of the partition. The next section shows how this information is organized so that it can be used and updated efficiently.

3. Representing the graph G(S*)

Forming the graph G(S*), defined in Section 1, can take $c \cdot v^2$ steps. However, this graph is determined by S, which is of length $O(c \cdot v)$, i.e., the graph has a short description. Using this description and the results of the previous section, we can improve the algorithm of Lewis.

We will build a balanced search tree from each clause of S. The clause $(x_1, \ldots, x_k)$ generates edges

$$\{(\bar{x}_i, x_j) \mid 1 \leqslant i, j \leqslant k, i \neq j\}$$

to $G(S^*)$. These edges will be represented by a tree T, whose leaves contain the literals $x_1, \ldots, x_k$. Thus, for each pair z, y of *distinct* literals in the leaves of T, $(\bar{z}, y)$ is an edge of $G(S^*)$. Each interior node of T contains a bit twocell(p), which indicates whether the subtree rooted at p contains at least two literals belonging to different cells, both alive. If this condition does not hold, then p contains the name of the cell to which all alive literals in p's subtree belong.

This representation makes it possible to find useful edges efficiently. For each literal z we maintain a list of all trees where the literal $\bar{z}$ occurs as a leaf; these trees represent edges of the form (z, w). Suppose we have in the depth-first search arrived at literal z. We choose a tree containing $\bar{z}$. We want to find a leaf with literal y, different from $\bar{z}$, such that $\text{cell}(y) \neq \text{cell}(z)$ and cell(y) has not been output yet. Then, (z, y) is a useful edge. The search for y can be done by starting upwards from $\bar{z}$ and using the information given by the twocell-bits. At an appropriate node the search turns downwards and proceeds to a leaf containing y. If such a leaf does not exist, then all the literals in the tree, except possibly $\bar{z}$, are either already output or belong to the same cell as z. Then the tree does not represent any useful edge starting from z. Therefore, another tree containing $\bar{z}$ is chosen and the process is repeated. This continues until a useful edge is found, or all trees containing $\bar{z}$ have been searched. In the latter case we know that no useful edge starting from z exists.

Next we estimate the time used for searching useful edges. The search takes $O(\log v)$ time per tree. Suppose z occurs in e_z trees. Then, finding one useful edge starting from z takes $O(e_z \cdot \log v)$ time. If we later have to find more useful edges starting from z, the search can be continued from where it was stopped previously, since an edge cannot become useful after it was useless. Thus, finding useful edges starting from z during the whole depth-first search can be done in $O(e_z \cdot \log v)$ time. As the sum of e_z's is bounded by $c \cdot v$, the total time needed for searching the useful edges is $O(c \cdot v \cdot \log v)$.

We still have to show how the trees can be updated efficiently. We maintain an array containing for each literal z the name of the cell in the partition currently containing z. For each cell, a linked list of its members is kept. When two components are merged during the search, we change the component names of the vertices in the smaller cell. (This is the standard weighted union rule for the union-find problem [6].) Therefore, each literal changes component at most log v times. When a literal z changes component, or the component containing z is output, we have to update the trees where z occurs. For one tree, $O(\log v)$ time is enough; to update all trees where z occurs takes $O(e_z \cdot \log v)$ time. Thus updating the information about z during the depth-first search can be done in total time $O(e_z \cdot (\log v)^2)$, and the updates for all variables can be done in $O(c \cdot v \cdot (\log v)^2)$ time.

4. Remarks

We have described an algorithm for renaming a set of clauses as a Horn set. The algorithm was based on the concept of a useful edge. Since there is only a linear number of such edges in any graph, following only them can speed up depth-first search. We showed that useful edges can be recognized on the basis of component information.

The graph arising in our application is a dense one, but it has a short description, the original formula. This description is used to build tree structures which make it possible to find useful edges reasonably fast. We suspect that a similar technique can be applied also to other classes of graphs with concise descriptions.

References

[1] A.V. Aho, J.E. Hopcroft and J.D. Ullman, The Design and Analysis of Computer Algorithms (Addison-Wesley, Reading, MA, 1974).

[2] B. Aspvall, M.F. Plass and R.E. Tarjan, A linear-time algorithm for testing the truth of certain quantified Boolean formulas, Inform. Process. Lett. 8 (3) (1979) 121–123.
[3] R. Fagin, Horn clauses and database dependencies, J. Assoc. Comput. Mach. 29 (4) (1982) 952–985.
[4] R. Kowalski, Predicate logic as programming language, in: J.L. Rosenfeld, ed., Information Processing '74 (North-Holland, Amsterdam, 1974) 569–574.
[5] H.R. Lewis, Renaming a set of clauses as a Horn set, J. Assoc. Comput. Mach. 25 (1) (1978) 134–135.
[6] K. Mehlhorn, Data Structures and Algorithms, Vol. 1: Sorting and Searching; Vol. 2: Graph Algorithms and NP-Completeness; Vol. 3: Multidimensional Searching and Computational Geometry (Springer, Berlin–Heidelberg–New York–Tokyo, 1984).