Visualising First-Order Proof Search

Visualising First-Order Proof Search

Graham Steel [1]

School of Informatics,
University of Edinburgh,
Edinburgh, EH8 9LE, Scotland

Abstract

This paper describes a method for visualising proof search in automatic resolution-style first-order theorem provers. The method has been implemented in a simple tool called *viz*, which takes advantage of the widely-supported scalar vector graphics format to produce graphs which can be viewed interactively. This allows the user to zoom in and out, pan, and get more information by clicking on particular parts of the graph. We demonstrate how the graphs can be used to suggest improvements to the strategy and heuristics used in the proof attempt.

Key words: First-order theorem proving, visualisation

1 Introduction

At first thought, it may seem that resolution-style first-order provers do not require elaborate user interfaces, since they are automatic 'push button' technologies. There is no choosing and invoking of tactics, as there is in an interactive prover. However, any serious effort to use a first-order prover to prove non-trivial theorems, or investigate open conjectures, involves considerable interaction. Typically, one first tries to prove smaller theorems and lemmas in the domain of interest. The behaviour of the prover is 'tuned' on these theorems, by trying different strategies and weighting functions, and by devising domain-specific redundancy rules. The prover can then be set to work on the final theorem. More interaction and changing of settings will typically be required before a final result is obtained. Even the experts employ this methodology: Larry Wos' recent book, [19], explains how it is used at Argonne labs, where the Otter team have successfully tackled several open questions in mathematics, including the famous Robbins conjecture.

One major difficulty in this interaction cycle is interpreting the output from the prover. Typically, this consists of lines of text scrolling rapidly up the screen, each containing details of a clause that has been derived. Even when a proof is eventually

[1] Email: graham.steel@ed.ac.uk

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

found, it is hard to see what went on during the search. This paper describes a simple tool, *viz*, for automatically producing trees illustrating the search pattern from this output. The trees include information about logical dependency, the weight of the clauses under consideration, the time elapsed in the search and, when a proof is found, the *critical path* of inferences leading to the proof. The graphs are produced as scalable vector graphics files, which can be viewed interactively by a variety of programs, allowing the user to zoom in and out, pan, and print selected areas of the graph. Clicking on a node causes an alert window to pop up giving the formula of the clause represented by that node.

In the rest of this paper, we first in §2 give a very brief introduction to automatic resolution-style theorem proving for those unfamiliar with the area. We describe our search trees and how they may be used in §3. We look at some related work in §4 and give further work and conclusions in §5.

2 Resolution-style Theorem Proving

Readers familiar with automatic first-order theorem proving may skip this section. For those requiring a more detailed treatment, one is provided in [13, Chapter 4].

Although many refinements have been made to the resolution method first proposed by Robinson, [12], the basic idea remains the same. Suppose you have a set of axioms, A, and a conjecture, C. Resolution-style provers proceed by negating the conjecture, and then attempting to derive a contradiction. All axioms and the conjecture are converted into *clauses*, i.e. formulae in clausal normal form. New consequences of the axioms and negated conjecture are derived by the resolution inference rule (or by an equational version, such as paramodulation). These new consequences are placed in the *usable set*. Proof search typically proceeds by picking a clause from the usable set, which becomes the *given clause*, and then applying resolution to it in all possible ways to derive new consequences. These new consequences are placed in the usable set, and the given clause is moved to the *worked-off set*. The process repeats until the empty clause is derived, indicating a contradiction.

The next given clause to be considered is often chosen by some weighting heuristic, giving a best-first search. A common heuristic is to chose the smallest clause (in terms of number of variables and function symbols) first. To get good results in a particular domain, quite complicated heuristics often have to be developed. The search process may also be modified in other ways, by changing the *strategy*. Commonly used strategies include looking out for derived clauses containing particular subterms (this is called *hotlisting*), or the so-called *set of support* strategy, where only a certain subset of the input clauses, and their consequences, are considered as possible given clauses. The strategy may also prescribe a depth-first or breadth-first search, ignoring heuristics. Additionally, the strategy may be altered at the inference level, by restrictions such as basicness, [1].

In modern provers, there are also a number of *reduction rules*, which operate on derived clauses. They either rewrite the clause to a smaller but equivalent ver-

sion, or attempt to detect the redundancy of the newly derived clause. A clause is redundant in a particular search if we have already derived a smaller equivalent, or more general version. Redundant clauses may be discarded without affecting completeness.

3 *viz* Search Trees

In this section we describe the trees produced by *viz*. Refer to Figure 1 for an example, where we show the search for a proof of Pelletier's 54th problem, [10], using the theorem prover SPASS, [18].

3.1 Tree Features

Search spaces for deductive problems are traditionally represented by trees. Each node on the tree represents a clause, and each edge represents logical dependency. However, in the specific case of first-order theorem proving, there is much more information we would like to be able to visualise. For example, many heuristics are based on the some kind of weight function over clauses. We would therefore like to be able to see the weight of each clause. In the trees produced by *viz*, this is represented by the size of the node. We scale all nodes with respect to the maximum weight reached during search, and a minimum value that represents a clause of weight zero.

Nodes are labelled with their clause number, and a second number denoting when the clause in question was considered as a given clause. Typically, the formula represented by the node would be far too long to be used as the node label. However, the tool would be cumbersome to use if the user had always to read off the clause number, and then refer manually to the raw output to see the actual content of the clause. Therefore, *viz* allows the user to click on a node, bringing up a pop-up window giving the clause itself and the numerical weight value. Figure 2 shows this in operation.

In the context of best-first search, we are interested in knowing in what order clauses were derived, and when they were considered as a given clause. This facilitates a search for bottlenecks, and heuristics that might overcome them. Our solution in *viz* is to colour the nodes in a progressive scale, for example from dark blue to white. Figure 1 illustrates this. The Darker clauses were considered first, the lighter clauses later. Axioms, which were never considered as given clauses, are white with a black outline.

An important feature of a successful search is the critical path, i.e. the chain of nodes leading to the derivation of the empty clause. This would be hard to see in a standard search tree, since the logical interdependencies of nodes quickly get complicated, even in the case of quite small proofs. Our system *viz* draws all nodes on the critical path in a diamond shape, allowing easy identification.

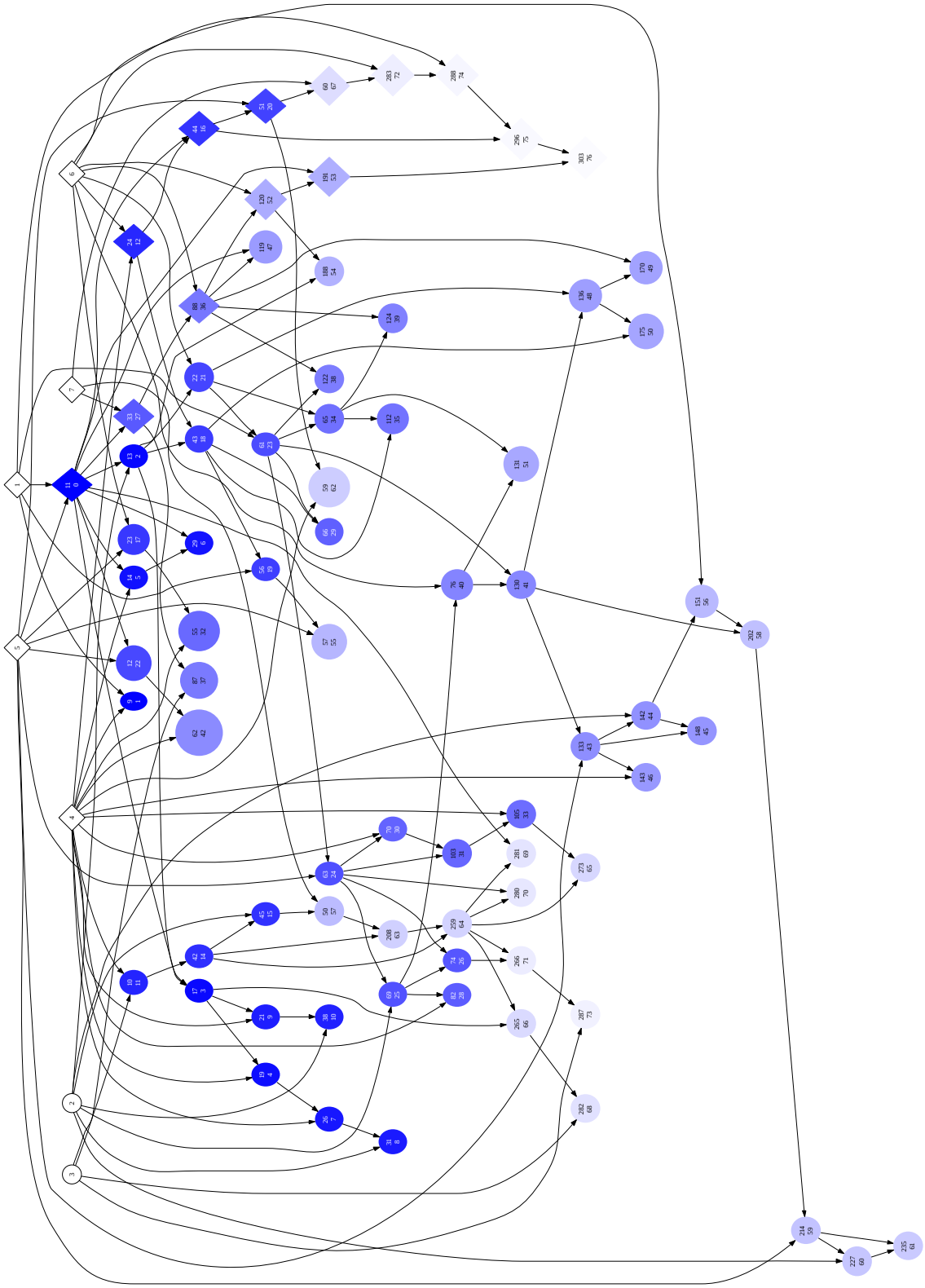

Fig. 1. Search tree: Pelletier's Problem No. 54

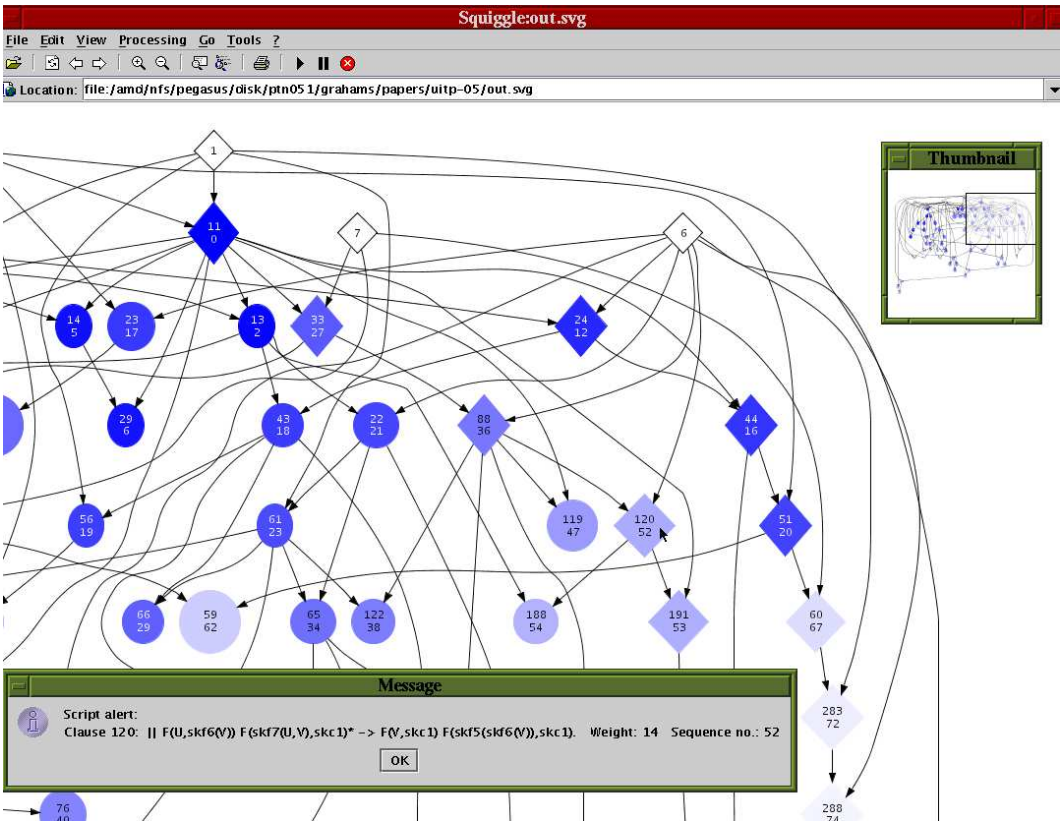

Fig. 2. Viewing a *viz* graph using Batik

3.2 Linear Search Trees

A particular strategy for resolution is the *linear strategy*. Here, new consequences are derived only by resolution between axioms and the given clause, and the first given clause must be the conjecture. No resolution steps between axioms are considered. The strategy is only complete for certain classes of problems, but this includes some significant applications.

The restriction of the strategy allows us to simplify the tree without losing important information. We do this by removing the axiom nodes from the tree, and instead labelling the edges with the number of the axiom that has been applied. Again, we allow the user to click on the edge label to see the axiom formula. This allows quite large proof searches to be visualised in a comprehensible fashion. Figure 3 gives an example of this. Here the search is for Gavin Lowe's attack on the Needham-Schroeder Public Key protocol, [7], as performed by CORAL, [14], a version of SPASS using a linear strategy.

The *viz* linear mode can also be used as simpli£cation technique for problems which do not in fact employ the linear strategy. The resulting graphs have far fewer edges and a simpler layout, emphasising the search pattern rather then logical dependence. Sometimes, however, this can make the proof unclear. A way around this would be to add back in all the edges for the critical path only. An example of this is given in Figure 4, where we have used *viz* to draw a linear simpli£cation

of the same search shown in Figure 1. The edges that have been added back have been coloured red.

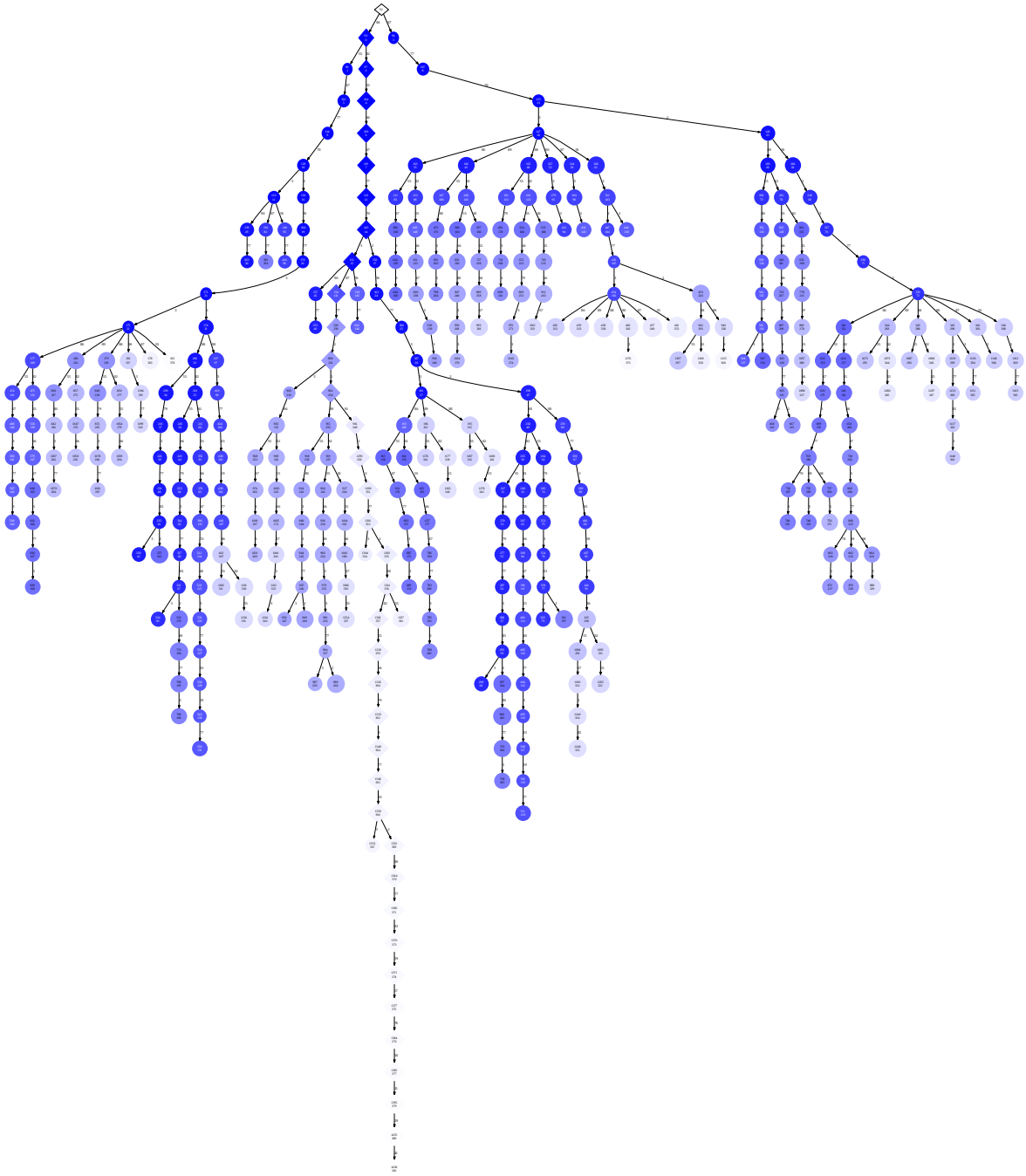

Fig. 3. Linear search tree - NSPK attack

3.3 Using the Trees

It is often possible to see a great deal even from a cursory inspection of the trees. For example, one can often see large ‘wasted areas’ - collections of nodes off the

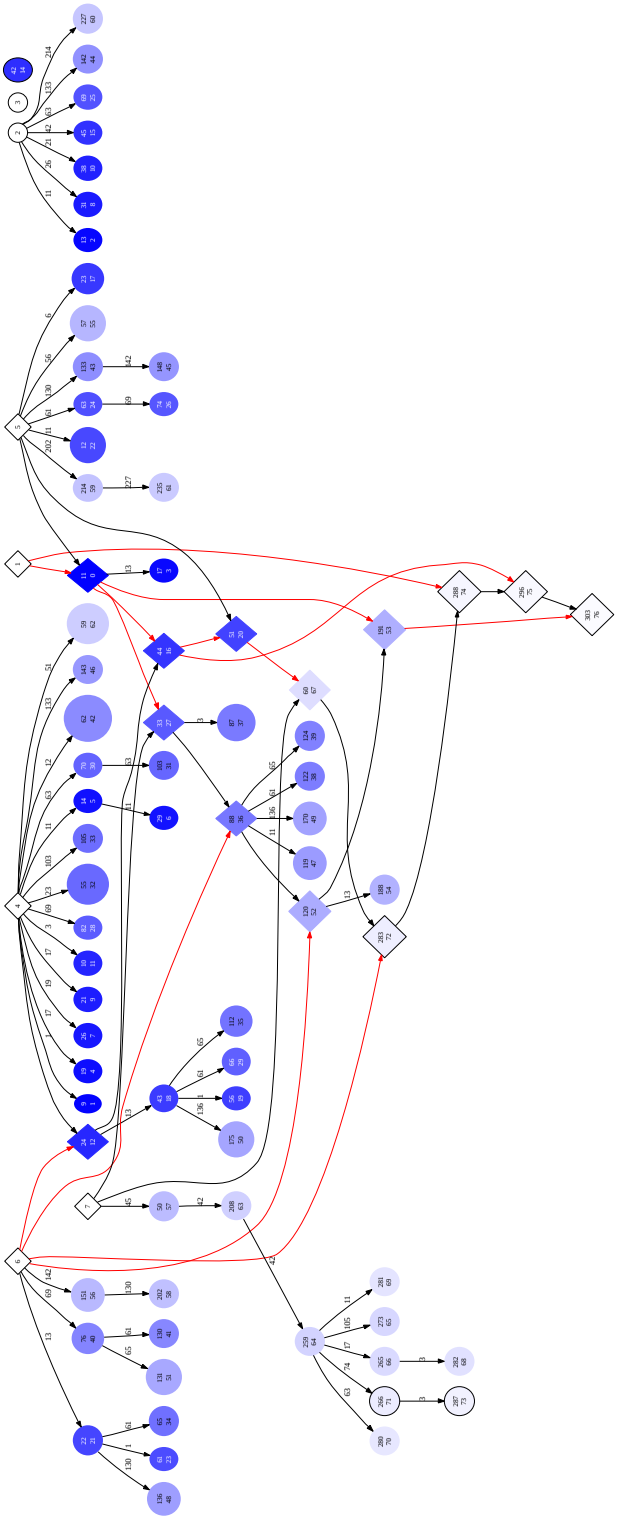

Fig. 4. Linear simpli£cation of Pelletier’s problem no. 54

critical path, that descend from just one node higher up the tree. This allows the user to inspect these higher nodes and think about whether there is a general rule (perhaps specific to the domain or theory) that could allow these nodes to be pruned from the search.

By tracing the colours of the nodes on the critical path, one can see where the discovery of the proof was 'held up'. If there is a marked change in colour from one node to another on the critical path, this indicates such a hold up in the proof discovery. By examining the clauses involved, and their weights as indicated by the node size, the user can improve the weighting heuristic. This may also suggest changes to strategy: for example, hotlisting a crucial subterm in the clause which holds up the critical path.

In Figure 5, we give another example of output from *viz*. This proof shows the discovery of a known attack on the Common Cryptographic Architecture API of the IBM 4758 hardware security module, [2]. In the fully zoomed-out diagram, we can see that there are two wide but shallow subsections of the proof, in the top left and bottom right portions of the diagram. Furthermore, we can see that all the clauses in the bottom right are descendants of one clause (marked on the diagram with an arrow). Examining this clause, we find that it requires our intruder to build a three part term using bitwise XOR to complete the attack. The three parts required are all in the initial knowledge of the intruder. However, the combinatorial possibilities of combining his knowledge with XOR are such that the majority of the proof search takes place just to put these three pieces together. We can see that this is where we might make an improvement in performance that will be vital for tackling larger problems.

We anticipate that other ways of using the graphs will suggest themselves, as our work with first-order provers in security protocol problems continues. We intend to continue the development of *viz*, adding more features to the graphs (see §5).

3.4 Implementation Details

viz first uses the 'dot' graph drawing program to produce an outline drawing of the graph. This is in scalable vector graphics (svg) format. A second pass is then made to add the annotations and pop-ups for the clause formulae. The final svg file can be viewed in any w3c compliant svg viewer. Batik [2], produced by the Apache organisation, is a freely available open-source svg viewer which seems to do a good job on the graphs *viz* produces.

At the moment, *viz* will only parse output from SPASS and daTac, [17], but it would be a simple matter to port it to other provers (a priority for future work is a version that parses the TSTP syntax, [16]). The *viz* script itself is available at `http://homepages.inf.ed.ac.uk/gsteel/viz/`.

[2] `http://xml.apache.org/batik/`

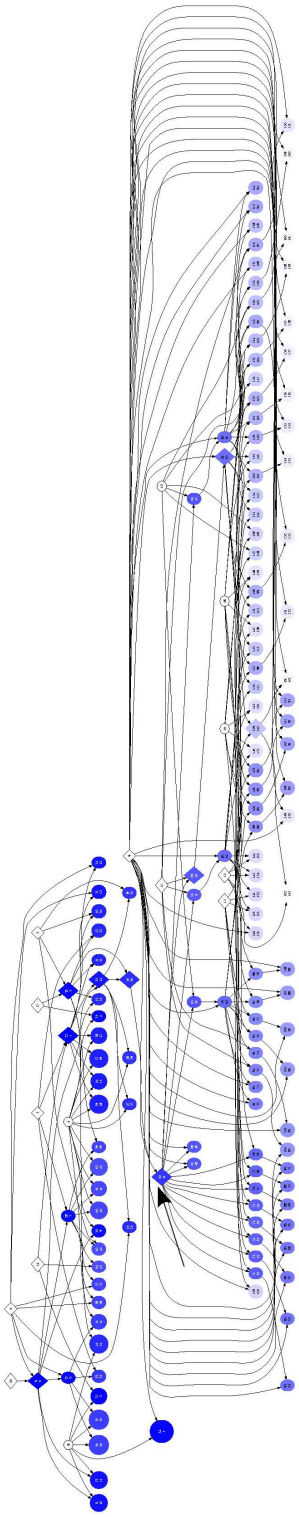

Fig. 5. Bond's attack on IBM 4758 CCA

4 Related work

In the past, trees have generally only been used in resolution theorem proving to illustrate a proof, rather than the search for a proof (e.g., the Tree Viewer for TSTP syntax proofs, [16]). However, search trees have been used in interactive theorem proving. Lowe and Duncan's XBarnacle, [8], used trees to facilitate user interaction with the proof planner *Clam*, [3]. XBarnacle allows the user to select the level of detail displayed, so the tree can be drawn with just nodes, or with nodes fully labelled with the subgoal at that node, and various levels in between. A further feature is to show the heuristic scores of all methods applicable to a node, to allow a user to see how the choice was made. We could perhaps adapt these features usefully to *viz*. Changing the level of detail could be achieved by some scripting commands which act on the svg file itself. We could use another pop-up window to offer information about the weighting score of other nodes in the usable set at the time each node was considered as a given clause. Usable sets are typically extremely large for non-trivial problems, so this would have to take the form of some kind of statistical summary.

Hutter and Sengler, [5], implemented a tree-based graphical user interface for the interactive prover, INKA, [6]. Their interface also used colour. Nodes were coloured to indicate if the subgoal had been proved, remained open, or was blocked (i.e. remained unsolved after all applicable methods had been applied). This is another idea we could use in *viz*. We could colour leaf nodes to indicate whether they had some children which remained in the usable set, or if all their children had been detected as redundant. This would be very useful information for the user.

5 Conclusions

Our tool *viz* seems to offer some immediate benefits in terms of visualising proof search in a resolution-style prover. These include being able to quickly identify bottlenecks and wasted areas. With further work, we should be able to offer more useful features to the user, such as marking leaf nodes with redundant children, and allowing the user to vary the level of detail on display. We also plan to adapt *viz* to other provers. It could then be used to compare different strategies on the same problems, as in [9], or to compare search performance on benchmark problems, for example the TPTP corpus, [15]. If adapted to draw the graphs in real-time, it could be used to track proof search during the CASC automated theorem proving competition, [11].

One problem we will certainly need to address is that of viewing large graphs. As we explained in §1, users of theorem provers typically train their settings on smaller examples which we can already visualise, but to gain real benefit from the tool, we would also have to be able to handle graphs with tens or hundreds of thousands of nodes. For the general problem of viewing large graphs, there has already been considerable work (see e.g. [4] for a survey). For our particular problem, we would probably also want to be able to do some pre-processing, in

order to get a summary picture of a large proof search. The use of the linear mode described in §3.2 is one example of how this could be done.

We intend to pursue all of these developments them as we continue our research into using first-order theorem provers to investigate computer security problems.

Acknowledgements

We are grateful to Geoff Sutcliffe for comments on an earlier draft of this paper.

References

[1] Bachmair, L., H. Ganzinger, C. Lynch and W. Snyder, *Basic paramodulation and superposition*, in: D. Kapur, editor, *11th Conference on Automated Deduction*, number 607 in LNCS, 1992, pp. 462–476.

[2] Bond, M. and R. Anderson, *API level attacks on embedded systems*, IEEE Computer Magazine (2001), pp. 67–75.

[3] Bundy, A., F. van Harmelen, C. Horn and A. Smaill, *The Oyster-Clam system*, in: M. E. Stickel, editor, *10th International Conference on Automated Deduction*, Lecture Notes in Arti£cial Intelligence **449** (1990), pp. 647–648, also available from Edinburgh as DAI Research Paper 507.

[4] Herman, I., G. Melançon and M. S. Marshall, *Graph visualization and navigation in information visualization: A survey*, IEEE Transactions on Visualization and Computer Graphics **6** (2000), pp. 24–43.

[5] Hutter, D. and C. Sengler, *The graphical user interface of INKA*, in: N. Merriam, editor, *Proceedings International Workshop on User Interfaces for Theorem Provers UITP-96*, York, U.K., 1996, pp. 43–50.

[6] Hutter, D. and C. Sengler, *INKA: the next generation*, in: M. A. McRobbie and J. K. Slaney, editors, *13th Conference on Automated Deduction* (1996), pp. 288–292, springer Lecture Notes in Arti£cial Intelligence No. 1104.

[7] Lowe, G., *An attack on the Needham-Schroeder public-key authentication protocol.*, Information Processing Letters **56** (1995), pp. 131–133.

[8] Lowe, H. and D. Duncan, *XBarnacle: Making theorem provers more accessible*, in: *14th Conference on Automated Deduction* (1997), pp. 404–408.

[9] McCune, W., *33 basic test problems: A practical evaluation of some paramodulation strategies*, in: R. Veroff, editor, *Automated Reasoning and its Applications: Essays in Honor of Larry Wos*, MIT Press, 1997 pp. 71–114.

[10] Pelletier, F. J., *Seventy-£ve problems for testing automatic theorem provers*, Journal of Automated Reasoning **2** (1986), pp. 191–216.

[11] Pelletier, F. J., G. Sutcliffe and C. Suttner, *The development of casc*, AI Communications **15** (2002), pp. 79–90.

[12] Robinson, J., *A machine-oriented logic based on the resolution principle*, Journal of the Association for Computing Machinery **12** (1965), pp. 23–41.

[13] Steel, G., "Discovering Attacks on Security Protocols by Refuting Incorrect Inductive Conjectures," Ph.D. thesis, University of Edinburgh (2004), electronic copy available on request from the author: graham.steel@ed.ac.uk.

[14] Steel, G., A. Bundy and M. Maidl, *Attacking a protocol for group key agreement by refuting incorrect inductive conjectures*, in: D. Basin and M. Rusinowitch, editors, *Proceedings of the International Joint Conference on Automated Reasoning*, number 3097 in Lecture Notes in Arti£cial Intelligence (2004), pp. 137–151.

[15] Sutcliffe, G. and C. Suttner, *The TPTP Problem Library: CNF Release v1.2.1*, Journal of Automated Reasoning **21** (1998), pp. 177–203.

[16] Sutcliffe, G., J. Zimmer and S. Schulz, *TSTP Data-Exchange Formats for Automated Theorem Proving Tools*, in: V. Sorge and W. Zhang, editors, *Distributed and Multi-Agent Reasoning*, Frontiers in Arti£cial Intelligence and Applications, IOS Press, 2004 (to appear).

[17] Vigneron, L., *Positive deduction modulo regular theories*, in: H. K. Büning, editor, *Computer Science Logic, 9th International Workshop*, Lecture Notes in Computer Science **1092** (1996), pp. 468–485.

[18] Weidenbach, C. et al., *System description:* SPASS *version 1.0.0*, in: H. Ganzinger, editor, *Automated Deduction – CADE-16, 16th International Conference on Automated Deduction*, LNAI 1632 (1999), pp. 378–382.

[19] Wos, L. and G. Pieper, "Automated Reasoning and the Discovery of Missing and Elegant Proofs," Rinton Press, 2003.