Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 226 (2009) 23–38
www.elsevier.com/locate/entcs

Visualizing Proof Search for Theorem Prover Development [1]

John Byrnes[2]

HNC Software
Fair Isaac Research
San Diego, California
johnbyrnes@fairisaac.com

Michael Buchanan Michael Ernst Philip Miller Chris Roberts Robert Keller[3]

Department of Computer Science
Harvey Mudd College
Claremont, California
{mbuchanan,mernst,pmiller,croberts,keller}@cs.hmc.edu

Abstract

We describe an interactive visualization tool for large natural deduction proof searches. The tool permits the display of a search as it progresses, as well as the proof tree itself. We discuss the feature set and architecture of the tool, including aspects of extensibility and the interface for interaction with other user-provided analysis and visualization code.

Keywords: Natural deduction, automated theorem prover, intercalation search, proof search visualization

[1] The authors are very grateful to our referees for supplying insightful questions, comments, and references which greatly improved the quality of this paper.
[2] This work was funded in part by the Intelligence Advanced Research Projects Activity (IARPA) Collaboration and Analyst System Effectiveness (CASE) Program, contract FA8750-06-C-0194 issued by Air Force Research Laboratory (AFRL). HNC Software is a subcontractor of the University of South Carolina in this contract. The views and conclusions are those of the authors, not of the US Government or its agencies.
[3] This project was conducted under the auspices of the Harvey Mudd College Clinic Program, with the sponsorship of Fair Isaac Corporation.

doi:10.1016/j.entcs.2008.12.095

1 Introduction

There are two main reasons why automated theorem proving in natural deduction particularly benefits from visualization. First, natural deduction is generally considered to be easier to read than most other logics (4; 15; 19); one motivation for doing theorem proving in natural deduction (ND) is to produce easily-comprehensible proofs, but in order to be understood they must first be put in an accessible form. Second, automated theorem proving (ATP) in natural deduction is still in early stages of study, and visualization provides a much-needed tool to assist in understanding the operation of experimental algorithms. By proving and visualizing in natural deduction, a single tool can display both human-readable proofs and be faithful enough to the data structures used by the reasoner to allow easy development of theorem proving algorithms.

Many ideas have been developed to aid in the comprehension of automatically generated proofs. Some we adopt for our purposes, such as graphical display (23). Others are unnecessary, such as converting proofs to natural deduction (15). Still others may aid proof comprehension but would do so at the cost of obscuring the function of the underlying theorem prover, such as conversion to natural language (6), and so we eschew them. We did not expect HTML-based browsers such as IWBrowser (of the Inference Web project, (13)) and SigmaKEE (of the SUMO project (17)) to facilitate high-level inspection of large proofs well. Interactive DAG viewers, such as the Interactive Derivation Viewer (25), are more likely to succeed at such a task.

In this paper we describe **ViPrS** (an acronym for **Visualizing Proof Search**), a tool for visualizing and interacting with proofs and partial proof search structures. It was designed with particular attention to use as an aid for proof search algorithm development and has several features to facilitate that use, including an extremely flexible programmatic user interface, the ability to interface directly with a theorem prover, and the ability to interact with the theorem prover as the search evolves step by step.

ViPrS was built specifically for the SILK theorem proving project, which we describe in section 2. The design is intended to be decoupled from SILK as much as possible, but effort was not spent in this initial version on allowing ViPrS to interact with arbitrary reasoning engines. SILK reads in proofs specified in a fairly simple XML format (described in (26)). Section 3 describes ViPrS, including its interface, implementation, and architecture. In section 4 we discuss experience with ViPrS to date, and possible future work is discussed in section 5.

2 The SILK Reasoning Project

Automated theorem provers have traditionally been directed toward problem solving in the domain of mathematics (24; 22). Although they have been successfully adapted to other formal domains, such as circuit verification, adaptation to informal domains of human knowledge has proven much more challenging. Large ontologies (18; 17) have been constructed to formalize reasoning in many domains, and the semantic web (2) offers the promise of ever growing amounts of formal knowledge over which software will attempt to reason. Traditional approaches to automated reasoning suffer from the combinatorial explosion of the search space that follows from the enormous number of concepts and axioms in large knowledgebases.

A number of extensions to traditional reasoning have been suggested. Proof planning (16) and strict segmentation of knowledge into microtheories (18) have shown promise. The approach of SILK, or "Soft Inference for Large Knowledgebases", is to adapt machine-learning techniques designed for unsupervised organization of unstructured data (primarily text) to the problem of structured knowledge. Patterns in the co-occurrence statistics of formal systems are exploited to automatically *compress* knowledge into a smaller vocabulary of higher-level concepts. Reasoning can occur much more efficiently at the higher level, and the resulting proofs can be used to guide proof search in the original vocabulary. In this regard, the high-level proof can be seen as a *plan*, and the approach is somewhat like proof planning with the exception that plans are discovered automatically in any domain rather than being coded from expert knowledge.

SILK also addresses the problem of reasoning under inconsistent assumptions. Large knowledgebases, especially those that grow organically such as the semantic web, are certain to occasionally introduce contradiction. Techniques which extract information from various data sources will introduce contradictions both due to extraction errors and due to the existence of inconsistent claims or erroneous entries in data sources. Because knowledge compression can also introduce contradiction, SILK needs to be especially robust. Classical theorem provers trivially reach arbitrary conclusions from contradictory assumptions, but SILK has the ability to prove relevant results without making arbitrary conclusions from inconsistency. This is achieved by using minimal logic via natural deduction proof search (this is an additional benefit to ND search over classical resolution search beyond the improved readability described in section 1). The reduced set of attainable conclusions is expected to be sufficient for many expected applications of "real world reasoning" (as opposed to theoretical mathematical reasoning), but this result needs to be established through usage. Of course SILK also has the option

merely to prefer minimal proofs, permitting classical inferences sparingly and perhaps notifying the user when doing so.

The technique of *intercalation* (21) has been adapted to create direct natural deduction search which is provably as efficient as search in the sequent calculus (4). SILK reasons directly in IKL (9), a dialect of Common Logic (CL) (5), which provides a very convenient and powerful syntax which looks higher-order but has a strictly first-order semantics over which it is sound and complete (8) [4]. KIF (the Knowledge Interchange Format (7)) is the most well known variant of CL. The typical approach to reasoning over knowledge represented in KIF by a first-order theorem prover is to use the "holds" translation into first-order logic, leading to computational and complexity difficulties (11). SILK attempts to overcome these difficulties through use of a natural deduction calculus in which reasoning is done directly in IKL without translation.

The intercalation theory underlying natural deduction search has been proven sound but has not received the large-scale implementation attention that has been given to resolution and other standard automated theorem proving techniques (12; 1). Reasoning directly in CL is novel, and the practical complexities are yet to be discovered. Reasoning in large knowledgebases of course yields very large search spaces and often large proofs as well, as inherited properties of classes must be established by reasoning through the subclass hierarchy. For all of these reasons, development of a practical, usable system such as SILK is likely to encounter many unforeseen obstacles which may be difficult to understand and overcome without tools such as ViPrS, which provide insights into the search patterns and inefficiencies that arise in practical application. SILK is in very early stages of development. It has not yet been reported on and is not available for public access.

3 ViPrS

SILK's data structure is particularly well-suited to graphical visualization. As described below, the structure is inherently non-linear, which makes displaying it textually unproductive. Instead, we visualize the search DAGs graphically. The objects to be displayed are also large enough that the interface must allow both inspection of the overall structure and of finer details, a challenge we address in several ways.

The search space for a proof is represented by a directed acyclic graph (DAG) rooted at the proposition which SILK is attempting to prove. *Proof search graphs* contain two distinct types of nodes. *Line nodes* represent logical

[4] Actually, this is true for the fragment of IKL implemented in SILK, which only allows finite expansion of row variables

formulas, while *rule nodes* represent rule applications. A rule node will have one line node conclusion and any number of line node premises. It is considered *proved* if all of its premises are proved. Likewise, a line node will have one or more *applications*, rule nodes for which it is a premise (except the root, which has none), and zero or more *justifications*, rule nodes for which it is the conclusion. A line node is considered *proved* if any of its justifications is proved or if it is an axiom or assumption.

A complete proof, then, consists of the proposition to be proved at the root, various internal nodes, and axioms and assumptions as leaves. A proof search graph is similar, except that its leaves may include formulas whose truth is as-yet undetermined. As such, proofs are merely a special class of proof search graphs, namely, completed ones. For conciseness we will henceforth refer to the object of visualization as a *proof search*.

3.1 Interface

A typical screenshot of ViPrS in use can be seen in figure 1. Here, we describe its salient features.

3.1.1 Viewing Pane

The main part of the ViPrS interface is the viewing pane. Here, the proof search is displayed in a traditional graphical format, with nodes represented as boxes and their relationships as lines between them. By default rule nodes have text indicating their type (which rule is being applied), while line nodes are blank (due to the lengthy propositions of most interesting proofs). The contents of a line node are viewable in a tooltip, or can be displayed using the command-line interface, as described in section 3.1.2.

The viewing pane allows various types of mouse-based interaction. Scrolling the mouse wheel zooms in and out of the proof, allowing inspection of the fine detail of proofs that are too large to easily fit on screen. Clicking and dragging pans the display. Clicking on a node selects it, causing it to visually increase in size and making it the object of future button presses. Hovering over a line node displays a tool-tip that contains its formula and the formulas that make up its context.

DAG layouts frequently attempt to minimize edge lengths and crossings by placing linked nodes near each other, but occasional long crossing edges are unavoidable. ViPrS simplifies the layout by treating the DAG in a very tree-like manner. The *primary parent* of each node is the parent closest to the root (note that the goal of the search provides a unique root to the search space), or leftmost in case of a tie. All other parents of a node connect to

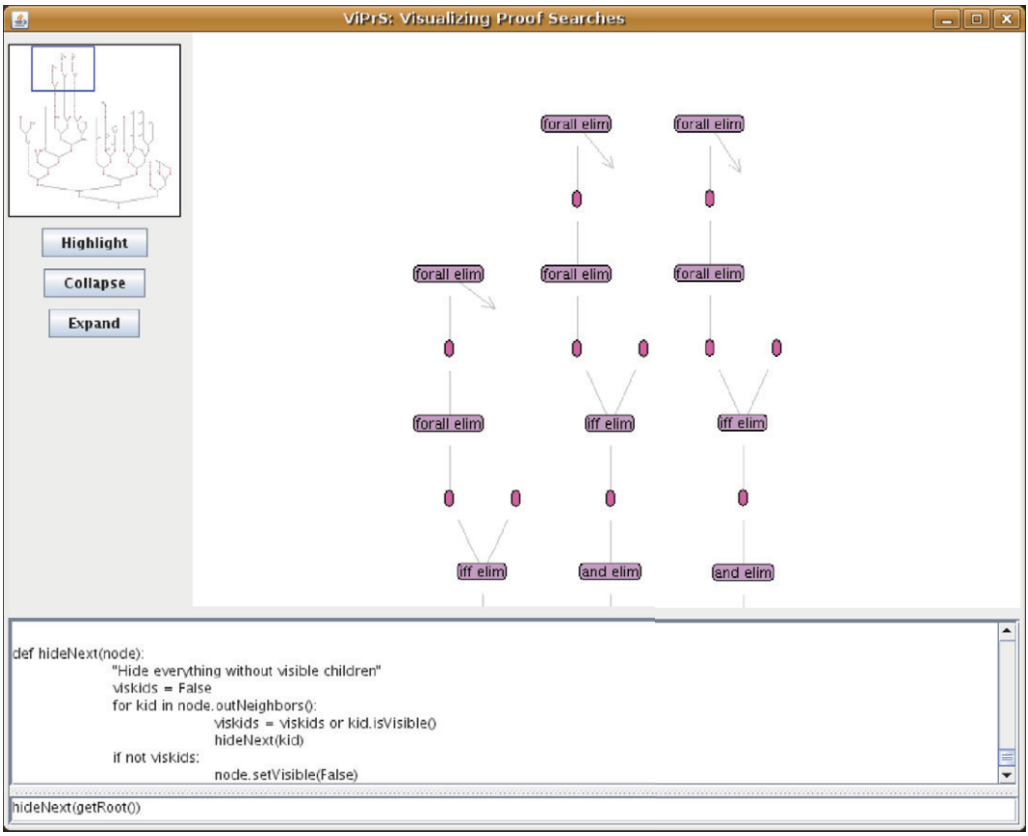

Fig. 1. A screenshot of the complete ViPrS system, with the main viewing pane in the center, the minimap and customizable buttons along the left, and the command-line interface at the bottom.

it via *crossing edges*, which are the short pairs of arrows displayed in figure 3. Clicking on one end of a crossing edge automatically pans the display to the far end of the edge and darkens both ends of the edge. The panning is most important when the far end of the edge is offscreen; the highlighting is especially useful when multiple crossing edge endpoints are visible.

3.1.2 Command Line Interface

Sited below the main viewing pane is the Command Line Interface (CLI), the primary means for the user to manipulate the proof search. The CLI lets the user interact programmatically with the visualization and the reasoning engine. Specifically, the user can enter arbitrary Python code and have it interpreted. Through predefined library functions and specially exposed variables, this code can interact with the visualization. This means that the user can query and manipulate every property of the proof search without the need for any explicit prior implementation of the particular interaction.

The exposed variables link not only to the visualization layer but into the data structures of the search algorithm as well. This means that any detail of the state of the algorithm can be inspected or adjusted at any step of search, allowing the user to understand and alter decisions made about backtracking, goal selection, etc. As the system developer changes the Java code in the reasoning engine, the new structures become exposed in ViPrS without any changes required to ViPrS so long as the original data structures can access the new structure.

The python library functions are loaded from a file at start up. The user can add arbitrary new function definitions to this file, expanding the library simply in python without editing any java code and without recompiling. The CLI also provides a *command history*. The command history makes it easy for the user to re-run previous commands, with or without modification. Our implementation lets the user both scroll through the history sequentially and search through it. The history persists between runs in a simple text file, allowing a user to return to commands from previous sessions. Function definitions entered during a session can be re-executed through the history mechanism or can be manually copied from the history file into the library file.

3.1.3 Dynamic Buttons

To the left of the viewing pane are a number of buttons to provide easy mouse-based manipulation of the proof search. As currently implemented, buttons act on the selected node(s). In the example configuration seen in figure 1, ViPrS has three buttons: one highlights a node by changing its color, and the others hide and display the sub-DAG rooted at the selected node.

The exciting aspect of the buttons is their easy customizability. Rather than running compiled-in code, each button is associated with a piece of Python code to run when it is clicked. A new button can be added by invoking a simple function, `addButton`, through the CLI. The buttons appearing on start up are defined similarly in the user's initialization script. Buttons can also be removed through an analogous `removeButton` function.

This sort of easy tool-building should be appreciated by and comfortable for our target audience of ATP developers and advanced users. In addition to saving us from having to anticipate the most frequently useful commands, dynamic buttons also hold distinctive advantages for users over buttons with fixed functionality. The user can create appropriate buttons to avoid switching back and forth between navigating a proof search with the mouse and manipulating it through the CLI. Making the buttons completely dynamic also saves users from having to recompile and restart, or even reload some

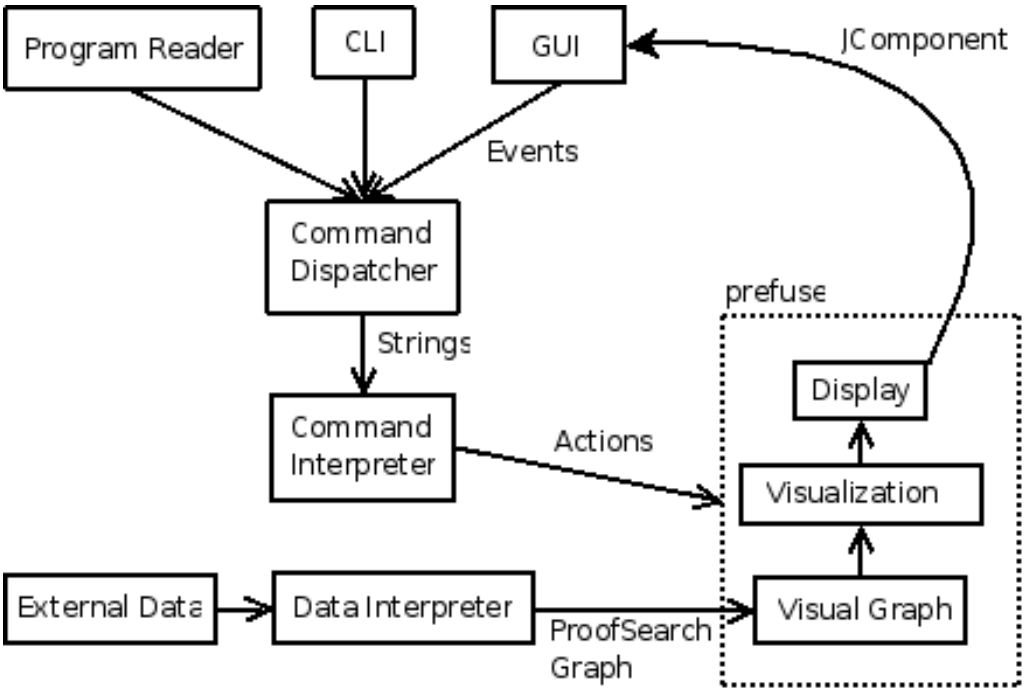

Fig. 2. The architecture of ViPrS, showing the data-flow relationships between the various components.

configuration file, to extend the functionality of ViPrS.

3.1.4 Minimap

Since proof searches can be so large, and the user may be focusing on only a small area at a time, we provide a summary view, or "minimap" of the proof search in a small box to the side. The minimap is a small, less-detailed view of the entire proof search. Its main utility is in showing what part of the proof search is currently in the viewing pane, by means of a rectangle surrounding that area. The minimap also allows the user easy navigation over large distances in the proof search by clicking on the area to be examined more closely.

3.2 Architecture

To support these features, the design of ViPrS needs to strike a balance between core features and the flexibility of a programmable system. Our architecture can been seen in figure 2. Extensibility was a driving factor in this arrangement of the functionality we have developed.

At the foundation layer, SILK provides the collection of nodes constituting

a proof search to be visualized. On top of this source data, the visualization component maintains an internal abstract model of the proof search graph.

From that abstract model, we derive a concrete visualization of the graph, with fields for all of the values salient to a visual display, including position, text, color, size, shape, and so forth. The main display and minimap are each a view onto this visualization. [5] These two displays, along with the other interface elements, are uniformly represented as Java Swing `JComponent` objects.

The command dispatcher responds to button presses and entries in the command line window by passing the code to be run to the command interpreter. The command interpreter, which holds references to the relevant objects of interest, evaluates the given code, and then asks the visualization system to update itself to reflect possible changes.

3.3 Implementation

3.3.1 Jython

Jython (3) provides the Python interpreter used as the back end of the CLI. It is a Python interpreter written entirely in Java, and allows interaction (such as function invocation and object reference) between compiled Java code and Python code in the interpreter.

3.3.2 Online Update

We use an observer pattern to let the visualization system register its interest in the changes presented by telling SILK to continue its search. This maintains the loose coupling between SILK and ViPrS. The observer object queues the changes presented by SILK, and then applies them to the visualizer's model of the search space at controlled points, where such changes won't upset the visualization. The user determines the number of search steps between updates, and can change this dynamically during a run.

The tree-like treatment of the DAG described in section 3.1.1 greatly simplifies this process. The primary parent completely determines the position of each node, so adding nodes to the DAG only spreads the display in the same way that adding nodes to a tree would do. Deletion of a node's primary parent without deletion of the node itself causes the primary parent to change, moving a subgraph of the display to a different region of the DAG. This is potentially more disruptive to the overall layout than the addition of nodes,

[5] This is a slight simplification–to improve the utility of the minimap, it actually keeps a separately derived visualization in which nodes are always displayed without text and at a fixed size.

but it still does not cause significant repositioning of many parts of the tree.

3.3.3 prefuse

The prefuse visualization toolkit [6] (10) is the open source software package used to drive the visual component of ViPrS. It is a software toolkit specifically designed for the visualization of graphs. It provides classes to model a graph with various visual characteristics, and then renders that model to a Java Swing `JComponent` which is embedded in the GUI.

Data being visualized by prefuse goes through a sequence of transformations, taking it from its raw, external form, through an internal abstract model, to an extension of the model to include concrete visual details, and finally rendering that visual model on a display. The SILK proof search data structures constitute the external form. From the graph implicit in this collection of objects, we create an explicit graph in an extension of prefuse's provided `Graph` class. We then augment this in a `VisualGraph` object that adds in details like layout, size, and color. Each rendered `Display` is a window onto that fully elaborated visualization of the graph structure, to which various interaction controls (e.g. pan, zoom) can be attached.

While prefuse provides various implementations of each of these transformation steps and interaction controls, they are often not precisely suited to the purposes of this visualization application. Thus, many of the implementations had to be tailored to our purposes. For example, the included tree-like layout didn't take account of its input being a rooted, directed graph.

4 Discussion

The ViPrS tool has proven itself to be very useful, even while in its developmental stage. First, in reporting research to sponsors, it was valuable to be able to display a visualization of a proof in order to explain its structure and some of the difficulties involved in proof search. Figure 3 is a ViPrS screen shot displaying a SILK proof constructed from an IKL translation of SUMO (the Suggested Upper Merged Ontology, (17)).

The tool has also been very useful for debugging of SILK. During proof search, certain heuristics depend on the height of given nodes. When part of the space did not seem to be getting explored, work in the debugger revealed a node for which the height was incorrectly recorded. Finding the root of this problem would have been extremely tedious through a standard debugger or logger, but a snapshot of the search space represented in ViPrS identified

[6] Per the conventions of its developers, prefuse's name is written in lower case

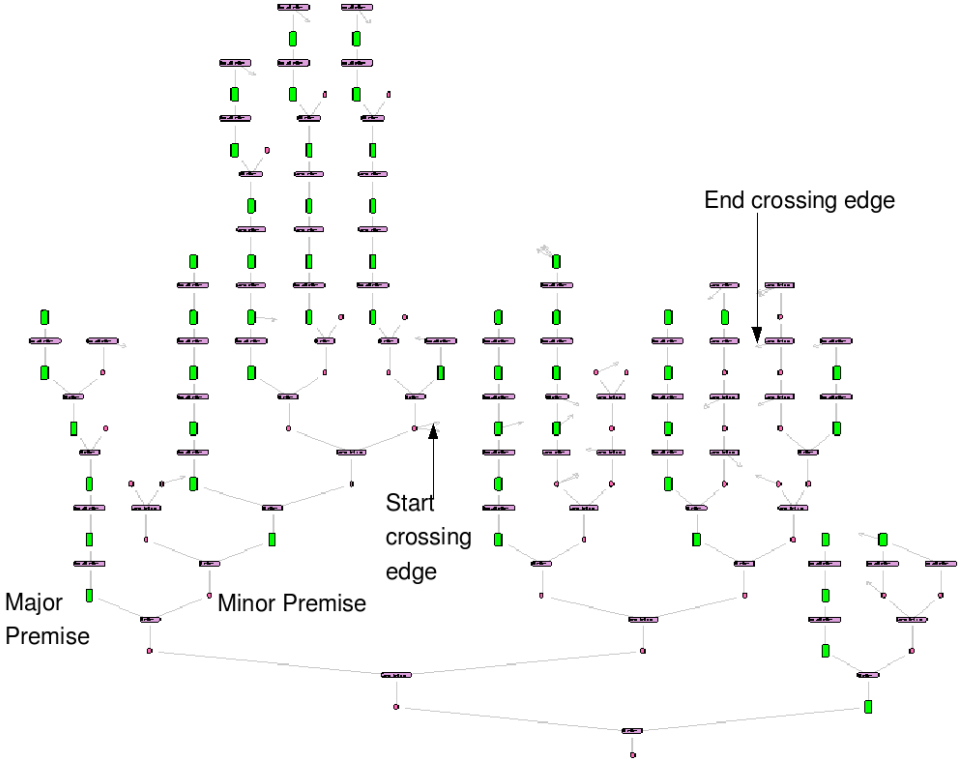

Fig. 3. Display of a simple proof. The darkest, horizontal nodes represent rule applications. The gray nodes are formula occurrences, and in this image the major premises have been highlighted. The restriction of major premises to the upper portion of each branch is a property of *normal* natural deductions (20; 4).

the roots of all mislabeled subtrees immediately. This was done without any change to the visualization source code. A simple recursive Python command was defined through the CLI which colored all nodes having height one green (simulated in figure 4). The nodes with the incorrect heights were immediately visible, and the state of each could be queried directly through the CLI. The visualization layer has no direct reference to the reasoning engine's node height values. Rather, the effect was generated by applying the ability described in section 3.1.2 of the CLI to access SILK data structures directly.

Beyond detection of bugs (coding errors), the real goal of the system is to understand the structure of the search space in order to improve search efficiency. The full search space can potentially contain redundant subtrees. These are recognized during search and treated specially so that redundant search does not occur. However, the use of Skolem functions and Herbrand terms during search introduces the possibility of parts of the search space which are redundant without being identical (rather, they are identical up to variable renaming). Proper treatment of these redundancies is best handled

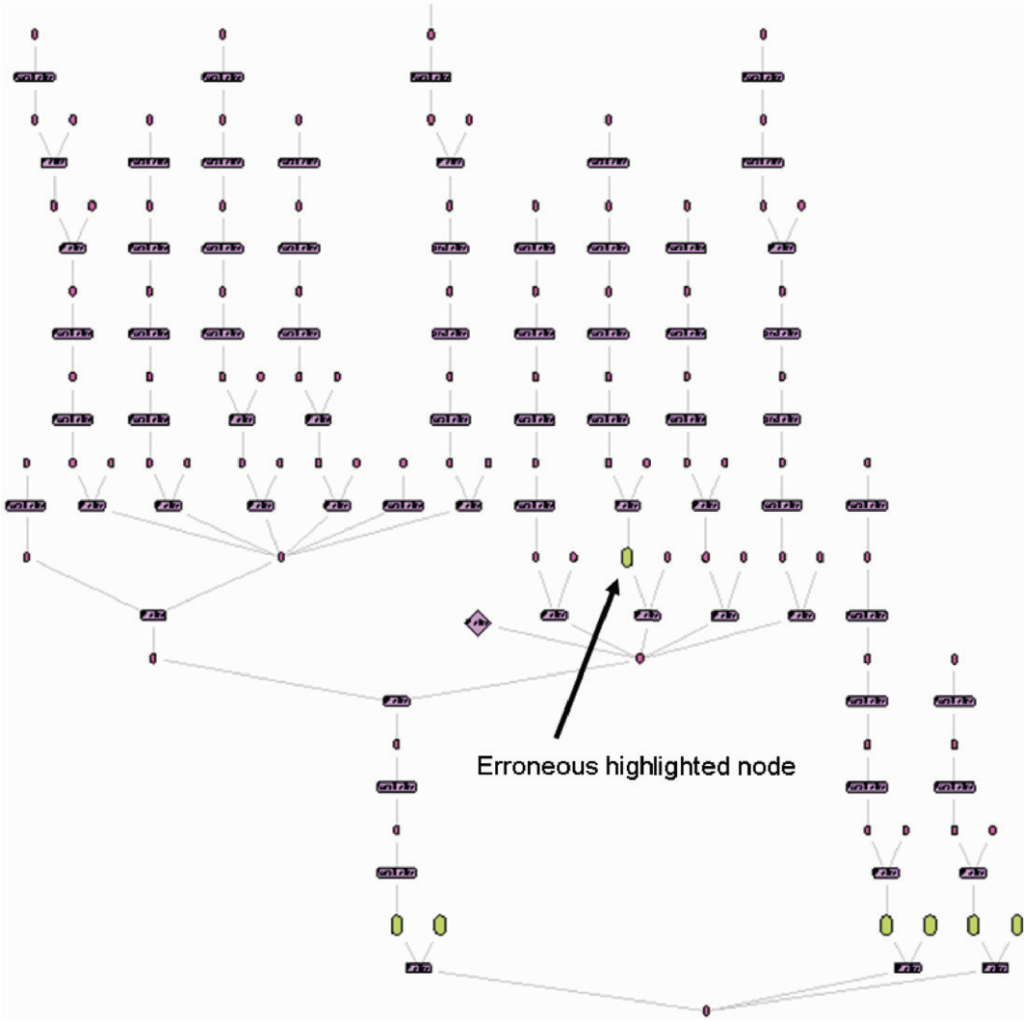

Fig. 4. The tree is searched and all nodes reporting height one are highlighted. The node with erroneous height stands out immediately. Traditional use of a debugger or text output would have required tedious comparison and manual inspection.

theoretically, but one often alternates between empirical algorithm exploration and theoretical development. When a particular proof search failed to yield a proof, the large scale visualization in figure 5 of part of the search space quickly suggested that redundant trees were being searched. Finer inspection then provided the determination that the redundancies seen were due to variable renaming and not due to coding errors with respect to the more straightforward type of redundancy.

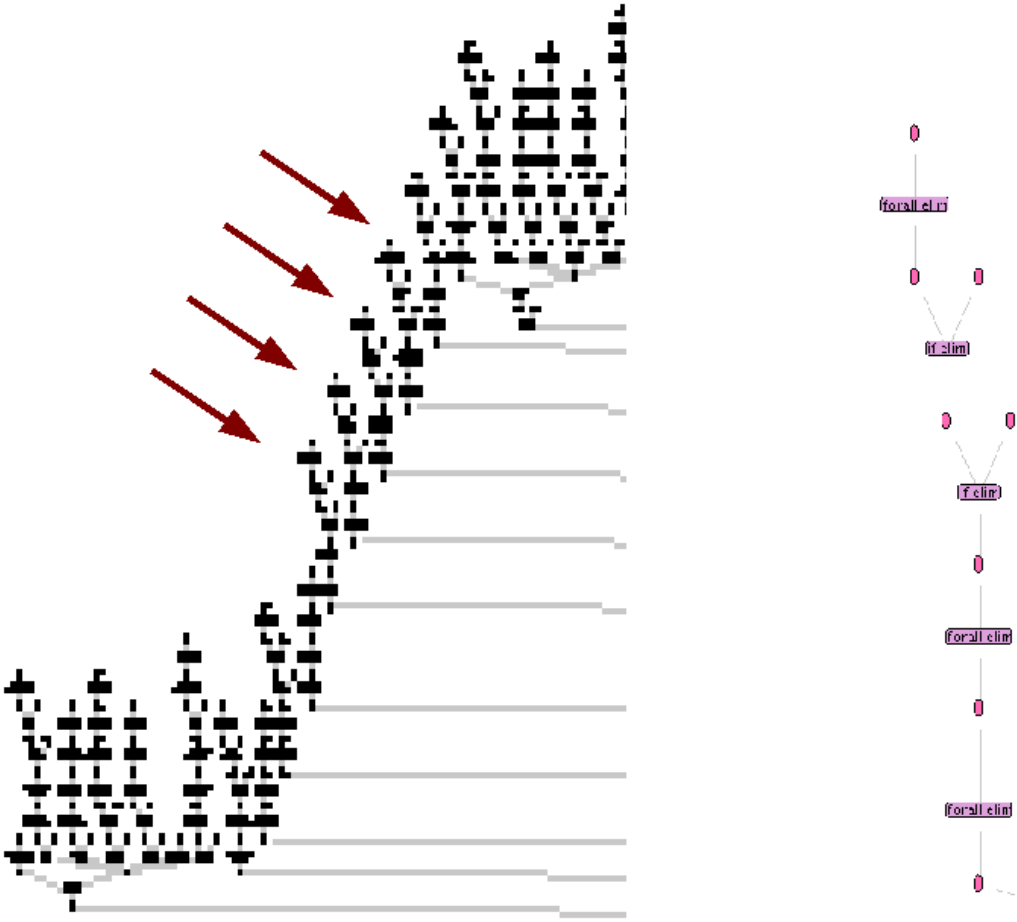

Fig. 5. The leftmost side of a large search tree after fifty steps of search. The close-up structure on the right appears at each of the subtrees indicated by the arrows. The tool-tip feature (described in section 3.1.1) immediately allows us to see that the root of each of these subtrees is identical up to choice of a newly introduced free variable.

5 Future Work

Extensibility was the primary consideration in the design of the ViPrS architecture. The simplest type of extension, as described above, is the addition of Python functions that can be called from the CLI. Colorings which highlight structural properties of the search space have been written already, such as those which highlight nodes that have been successfully proven, or those which are major premises (as the ND search is driven by restriction to *normal* deductions).

One important accomplishment is the decoupling of the minimap from the primary display. Although the two diagrams currently represent the same view at different scales, the minimap can render a different view entirely. One

possibility is embedding nodes into points on the plane without rendering individual nodes or edges. The points of the plane could be colored to represent properties of nodes, such as a heat map representing the ages of nodes to indicate the order in which parts of a tree were visited. Other maps might represent the number of free variables occurring in each node, or the number of instances of members a given set of formal symbols (names) from the knowledgebase that might be of particular interest to a user.

Collapsing of nodes is crucial to readability of the graphs, but the appropriate mechanisms for collapsing DAGs are not exactly clear. The current implementation hides as much as possible, meaning that all nodes "above" a collapsed node are hidden, even if they are above through crossing edges (the arrows) rather than direct edges; as a result collapsing may hide distant nodes unexpectedly. Convenient techniques for allowing the user to control this functionality and for indicating points which have been collapsed remotely need to be developed.

One of the motivations of ND theorem proving is to provide a more human-readable proof. The fact that intercalation always finds normal proofs can be exploited to automatically collapse parts of the visualization based on minimal nodes and the branch structure of normal derivations. The plans which SILK generates to provide guidance for proof search should give guidance for collapsing as well, and future work should provide the ability to present a plan which expands to the underlying proof.

Further extensions of SILK will likely also lead to extensions of the visualization. SILK is currently being extended to interoperate with BRUSE, a Bayesian network software system which provides soft evidential updating (26; 27). SILK proofs are being converted into network fragments as a means of automating network construction, and the ViPrS system is likely to be extended to provide visualization of the resulting Bayesian networks.

An appealing direction for ViPrS not originally considered is to allow arbitrary reasoning engines to make use of ViPrS for visualization. The current system with SILK could be used to read in arbitrary proofs and proof search DAGs specified in SILK's XML format, so an easy extension that might be useful is simply to let SILK read other standard formats such as TPTP (24) and PML (14). Of potentially greater utility is development of a Java API which developers of reasoning engines could use in order to provide interactivity with any algorithms being developed. This would allow, for example, different reasoners (possibly using different logical calculi) to be run in parallel in order to compare their approaches to various problems of interest.

References

[1] Beckert, B. and J. Posegga, *leanTAP: Lean tableau-based deduction*, Journal of Automated Reasoning **15** (1995), pp. 339–358.
URL http://citeseer.ist.psu.edu/beckert95leantap.html

[2] Berners-Lee, T., J. Hendler and O. Lassila, *The semantic web*, Scientific American (2001).

[3] Bock, F., *Jython project* (2007).
URL http://www.jython.org/

[4] Byrnes, J., "Proof Search and Normal Forms in Natural Deduction," Ph.D. thesis, Department of Philosophy, Carnegie Mellon University (1999).

[5] Delugach, H., *Common logic - a framework for a family of logic-based languages*, International Standards Organization Final Committee Draft (2005).
URL http://cl.tamu.edu/docs/cl/32N1377T-FCD24707.pdf

[6] Fiedler, A., *P.rex: An interactive proof explainer*, in: R. Gore, A. Leitsh and T. Nipkow, editors, *Proceedings of the International Joint Conference on Automated Reasoning*, number 2083 in Lecture Notes in Artifical Intelligence (2001), pp. 416–420.

[7] Genesereth, M. R. et al., *Knowledge interchange format*, draft American National Standard (1998).
URL http://www.ksl.stanford.edu/knowledge-sharing/kif/

[8] Hayes, P. and C. Menzel, *A semantics for the knowledge interchange format*, in: *Proc of IJCAI 2001 Workshop on the IEEE Upper Ontology*, 2001.
URL http://reliant.teknowledge.com/IJCAI01/HayesMenzel-SKIF-IJCAI2001.pdf

[9] Hayes, P. and C. Menzel, *IKL specification document* (2006).
URL http://www.ihmc.us/users/phayes/IKL/SPEC/SPEC.html

[10] Heer, J., S. K. Card and J. A. Landay, *prefuse: a toolkit for interactive information visualization*, in: *CHI '05: Proceedings of the SIGCHI conference on Human factors in computing systems* (2005), pp. 421–430.

[11] Horrocks, I. and A. Voronkov, *Reasoning support for expressive ontology languages using a theorem prover*, in: *Foundations of Information and Knowledge Systems (FoIKS)*, 2006.

[12] McCune, W., *Prover9* (2008).
URL http://www.cs.unm.edu/~mccune/prover9/

[13] McGuinness, D. L. and P. P. da Silva, *Explaining answers from the semantic web: The inference web approach*, Journal of Web Semantics **1** (2004), pp. 397–413.
URL http://browser.inference-web.org/

[14] McGuinness, D. L., L. Ding, P. P. da Silva and C. Chang, *PML 2: A modular explanation interlingua*, in: *AAAI 2007 Workshop on Explanation-aware Computing*, Vancouver, British Columbia, Canada, 2007.

[15] Meier, A., *System description: Tramp - transformation of machine-found proofs into natural deduction proofs at the assertion level*, in: D. McAllester, editor, *Proceedings of the 17th Internal Conference on Automated Deduction*, number 1831 in Lecture Notes in Artifical Intelligence (2000), pp. 460–464.

[16] Melis, E. and A. Bundy, *Planning and proof planning*, in: S. Biundo, editor, *ECAI-96 Workshop on Cross-Fertilization in Planning*, Budapest, 1996, pp. 37–40.

[17] Niles, I. and A. Pease, *Towards a standard upper ontology*, in: *FOIS '01: Proceedings of the international conference on Formal Ontology in Information Systems* (2001), pp. 2–9.

[18] Panton, K., C. Matuszek, D. Lenat, D. Schneider, M. Witbrock, N. Siegel and B. Shepard., *Common sense reasoning—from cyc to intelligent assistant*, in: Y. Cai and J. Abascal, editors, *Ambient Intelligence in Everyday Life*, number 3864 in LNAI, Springer, 2006 pp. 1–31.

[19] Pfenning, F., "Proof Transformations in Higher-Order Logic," Ph.D. thesis, Carnegie Mellon University, Pittsburgh (1987).

[20] Prawitz, D., "Natural Deduction: A Proof-Theoretic Study," Dover Publications, 2006.

[21] Sieg, W. and R. Scheines, *Searching for proofs (in sentential logic)*, in: L. Burkholder, editor, *Philosophy and the Computer*, Westview Press, Boulder, 1992 pp. 137–159.

[22] Siekmann, J., C. Benzmüller, V. Brezhnev, L. Cheikhrouhou, A. Fiedler, A. Franke, H. Horacek, M. Kohlhase, A. Meier, E. Melis, M. Moschner, I. Normann, M. Pollet, V. Sorge, C. Ullrich, C. Wirth and J. Zimmer, *Proof development with omega*, in: A. Voronkov, editor, *Proc. CADE-18*, number 2392 in LNAI (2002).

[23] Steel, G., *Visualising first-order proof search*, in: D. L. C. Aspinall, editor, *Proceedings of User Interfaces for Theorem Provers 2005*, 2005, pp. 179–189.

[24] Sutcliffe, G. and C. Suttner, *The tptp problem library: Cnf release v1.2.1*, Journal of Automated Reasoning **21** (1998), pp. 177–203.

[25] Trac, S., Y. Puzis and G. Sutcliffe, *An interactive derivation viewer*, in: *Proceedings of the 7th Workshop on User Interfaces for Theorem Provers (UITP-06)*, Electronic Notes in Theoretical Computer Science **174**, 2007, pp. 109–123.

[26] Valtorta, M., J. Byrnes and M. Huhns, *Logical and probabilistic reasoning to support information analysis in uncertain domains*, in: *Proceedings of the Third Workshop on Combining Probabilty and Logic (Progic-07)*, Canterbury, England, 2007.

[27] Valtorta, M., Y.-G. Kim and J. Vomlel, *Soft evidential update for probabilistic multiagent systems*, International Journal of Approximate Reasoning **29** (2002), pp. 71–106.