Visualising Reasoning: What ATP can learn from CP

John Slaney[1,2]

Australian National University
and National ICT Australia

Abstract

Tools for graphical representation of problems in automated deduction or of proof searches are rare and mostly primitive. By contrast, there is a more substantial history of work in the constraint programming community on information visualisation techniques for helping programmers and end users to understand problems, searches and solutions. Here we consider the extent to which concepts and tools from a constraint programming platform can be adapted for use with automatic theorem provers.

Keywords: automated reasoning, visualisation

1 Introduction

There is a growing appreciation in the automated theorem proving community of the importance of the interface between automated reasoning technology and its users. In the present paper, we wish to contribute some ideas concerning one aspect of this interface: the visualisation of proof problems and proof searches. We draw on our experience in developing the user interface and visualisation toolkit of the constraint programming platform G12 [8]. The application of information visualisation techniques is more advanced in the constraint programming (CP) and operations research (OR) communities than in that of automated deduction, mainly because of a long history of use of the systems by third party programmers to provide customised solutions

[1] NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council through the ICT Centre of Excellence program.

[2] Email: mailto:john.slaney@anu.edu.au

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

for end users [4]. We limit ourselves to visualisation of logical problems and of search by "bottom-up" theorem provers based on resolution or superposition, for instance. In this paper we do not consider graphical presentation of *proofs*. This is not to deny the importance of proof presentation, but it is not the area in which CP visualisation ideas are most directly applicable.

1.1 Constraints and logic

Constraint satisfaction problems (CSPs) are classically presented in terms of a finite set of *decision variables* $v_1, \ldots, v_n$ with associated *domains* or sets of possible values $D_1, \ldots, D_n$, and a set of *constraints* or relations between the variables. That is, a constraint on a tuple of variables $\langle v_{i_1}, \ldots, v_{i_k} \rangle$ is a subset of $D_{i_1} \times \ldots \times D_{i_k}$. An *assignment* is a function assigning to each decision variable v_i a value from D_i. Assignment $\mathcal{A}$ satisfies a constraint $\mathcal{C}$ on $\langle v_{i_1}, \ldots, v_{i_k} \rangle$ iff $\langle \mathcal{A}(v_{i_1}), \ldots, \mathcal{A}(v_{i_k}) \rangle \in \mathcal{C}$. $\mathcal{A}$ is a solution to the CSP iff it satisfies all of the constraints. Frequently, what is sought is not merely an arbitrary solution but an optimal one, defined as one minimising the value of some particular decision variable caled the *objective*.

In terms of first order logic, domains of the decision variables are domains in the ordinary model-theoretic sense. If the domains are not all the same, then the logic is best construed as many-sorted. Decision "variables" are not variables in the true sense, as they cannot be bound by quantfiers for instance, but are individual constants or zero-ary function symbols. An assignment is an interpretation of the language, just as in the model theory of first order logic, and a constraint is a relation which might well be expressed as a formula which is true or false on the interpretation in the expected way.

G12, like other constraint programming systems, provides a modelling language. This language, Zinc [5], is used to state problems declaratively and independently of the algorithms used to solve them. Zinc is expressly designed as a notation for the first order logic of (mainly) finite domains, with additional apparatus for dealing with numbers. It has a type structure, with basic types `int`, `float`, `bool` and user-defined enumerated types. To these are applied the type constructors 'array of' and 'set of' as well as formation of subrange types and record types (type tuples). Quantifiers `forall` and `exists` can range over any finite set, and the boolean connectives are available. Despite the potentially intricate types, many CSPs are expressed using little more than base types and arrays of them. On the construal of Zinc as logic, arrays are simply functions from their index set to their value set, so array names and indices are function symbols and arguments. Details of the syntax of Zinc are not germane to the present document: it suffices that it is a way of writing logical theories and that several visualisation tools have been developed for it.

1.2 Example 1: Meet-pass planning

```
set of int: Sectors = 1..6;
set of int: Trains = 1..3;
set of int: Steps = 1..13;
array[Sectors,Sectors] of bool: linked;
array[Trains] of Sectors: start;
array[Trains] of Sectors: finish;
array[Steps,Trains] of var Sectors: pos :: is_output;

constraint forall(t1, t2 in Trains, x in Steps)
  (pos[x,t1] == pos[x,t2] -> t1 == t2);
constraint forall(t in Trains)
  (pos[1,t] == start[t] /\ pos[nsteps,t] == finish[t]);
constraint forall(t in Trains, x in Steps where x > 1)
  (linked[pos[x,t],pos[x-1,t]]);
constraint forall(t,u in Trains where t != u)
  (forall(x in Steps where x > 1)(pos[x-1,u] != pos[x,t]));

start  = [ 1, 2, 4 ];
finish = [ 5, 2, 4 ];
linked = [| true,  true,  false, false, false, false
          | true,  true,  true,  false, false, false
          | false, true,  true,  true,  false, true
          | false, false, true,  true,  true,  false
          | false, false, false, true,  true,  false
          | false, false, true,  false, false, true |];

solve satisfy;
```

Fig. 1. Meet-pass planning problem in MiniZinc

To illustrate how CSPs are formulated and visualised, we consider a problem in meet-pass planning. Industrial meet-pass solvers, of course, routinely solve much larger instances than this, but it serves the present purpose.

Description

Five sectors of railway track are connected linarly from S1 to S5. There are trains in sectors S1, S2 and S4. There is a siding accessible from sector S3 big enough to hold one train. At each timestep, each train may stay where it is or move into an empty adjacent sector. No two trains may be in a sector at the same time. Find a plan of 13 timesteps which moves the train in S1 to S5, returning the other two trains to their starting positions in S2 and S4.

2 Views

Before considering the potential of visualisation for aiding understanding of first order theorem proving, we outline some of the ways in which it is routinely applied in constraint programming. Our examples come from the visualisation toolkit of the G12 constraint programming platform, since it is our own work, so we know it best.

2.1 Constraint graphs

The *constraint graph* of a CSP is the undirected graph whose vertices are the decision variables of the problem and whose edges connect variables which co-occur in at least one constraint. Inessential variants may weight the edges according to the strength of the constraints, or according to the number of constraints relating the variable pair, or may have additional vertices for problem parameters or data values. Also sometimes useful are the *inverse constraint graph*, in which the vertices are constraints and the edges represent sharing of variables, and the *bipartite constraint graph* which has vertices of both types (decision variables and constraints) and arcs for the 'occurs in' relation. The static view of the constraint graph shows the "shape" of the problem. It is especially good for revealing clustering of variables, critical variables through which a lot of information must flow during propagation or during variable choice, certain kinds of symmetry or repeated structure, and disconnected or loosely connected fragments of the problem. Hence the constraint graph is a picture of *structure*.

Note that the constraint graph represents only the existence of constraints, not their content. This level of abstraction can reveal which parts of a problem are closely related and which are remote. It shows the pathways along which information can be passed by propagators associated with constraints, but does not show *what* information may be passed or in which direction. Importantly, it is completely independent of whether the problem is satisfiable or not.

During search, the constraint graph may be viewed in dynamic mode, showing at each step which variables have values assigned and which domains have been reduced and which have not. This animated picture can show the search concentrating on a particular part of the problem before leaping to another, or making scattered decisions all over it. For the constraint programmer, this helps to form a mental image of the search in progress which is part of understanding the effect of variable choice heuristics, channeling constraints and much else.

A constraint graph viewer needs a visual metaphor, a layout definition with associated layout algorithm, access to the problem syntax before and during search, and a rich set of controls. The standard visual model is the

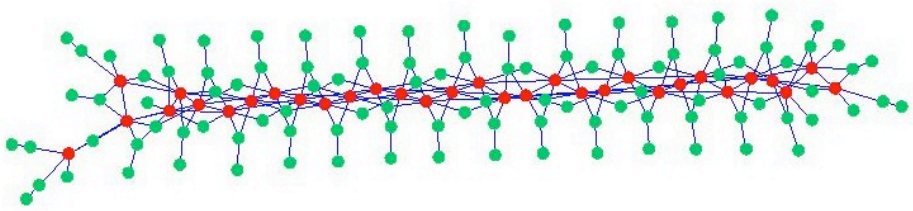

Fig. 2. Meet-pass planning problem constraint graph. The long shape is typical of planning problems. The array 'pos' is in red, the auxiliary variables in green.

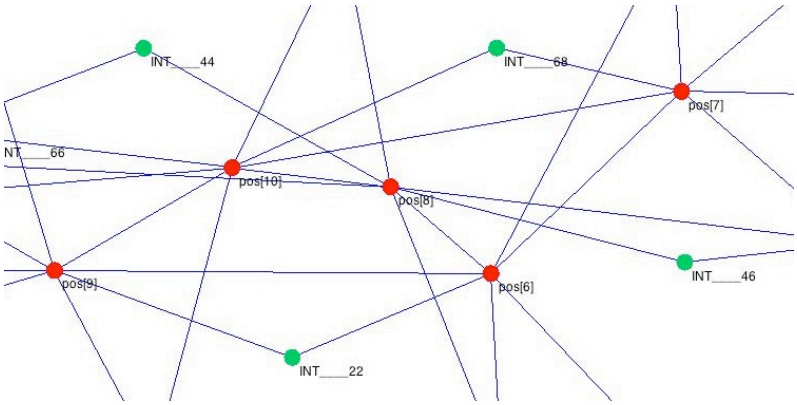

Fig. 3. Detail from the graph in Figure 2

"balls and sticks" metaphor. Vertices are shown as circles or spheres, and edges as straight lines, sometimes with meaningful thickness. Some viewers allow curved lines and some allow rectangles or other shapes for the vertices, usually in order to allow text to be displayed inside them. The G12 constraint graph viewer, G12-CGV, uses simple circles and straight lines. Its layout is force-directed [6]: edges exert on the vertices they join an attraction directly proportional to their length, like springs with a rest length of zero. At the same time, all vertices repel each other with a force obeying an inverse-square law, as though they were similarly charged particles. This type of layout is good for revealing clustering and symmetry, but can result in cluttered graphs in some cases in comparison with layouts which, for instance, aim for regular spacing of vertices or try to minimise edge crossings. The layouts generated by G12-CGV are 2-dimensional, in order to reduce overlap and masking in static pictures on a screen or page. 3-dimensional pictures are sometimes useful (and undeniably pretty) but are less easy to work with in general.

2.2 Search trees

Constraint satisfaction problems may be approached by many search methods each with a great range of variants. The fundamental division is between local search methods such as hill-climbing, tabu search, random walks, simulated annealing or the like, and systematic search methods based on some notion of traversing a tree. Tree-based searches standardly proceed in a roughly similar way: at each node of the search tree, a decision variable is chosen and its domain split. The effects of this decision are propagated through the constraints, meaning that some inference rules are woken and run to a fixed point. This may remove values from the domains of other variables, possibly triggering assignments where domains are reduced to singletons and possibly causing the search to backtrack if a domain is emptied or "wiped out". The resulting subtrees are explored recursively.

Visualisation toolsets for constraint programming [2,1,3] invariably feature representations of search trees. G12 is no exception, providing a simple search tree viewer G12-STV. Trees may be viewed statically after the event or dynamically growing during the search. Viewing the entire tree gives an impression of the nature of the search: are the branches getting longer or shorter as the search progresses? Are there repeated subtrees of the same shape, indicating repeated work? Where in the search does a particular decision fit? Zooming in on a specific node or a small set of nodes allows individual decisions to be examined in detail. Drawing a tree is rather easy, and mechanisms for popping up information on request are readily available in graphics libraries.

Some modifications of the basic binary tree are easy to incorporate in the view: multi-way branching, for instance, poses no problem. More significant departures from the simple paradigm, however, pose challenges for visualisation. SAT solvers, for instance, frequently perform sophisticated backjumping, relying on learned clauses to ensure progress through a virtual tree which changes constantly; the problem of representing this process in a usable search tree viewer is largely open. Visualising more radically different methods such as local search, is another game: there is no obvious analogue of the search tree view presenting such a combination of global search shape and local reasoning detail.

2.3 Custom views

Having solved a CSP, it is usually necessary to present the solution to a user in some readily understandable way, and for this purpose graphical representations are commonplace. There is no practical limit to the range of pictures that might be employed. A few, like Gantt charts, histograms, network diagrams or dots on maps are common enough to count as standard, but much more eleborate possibilities exist. The solution to a problem in product con-

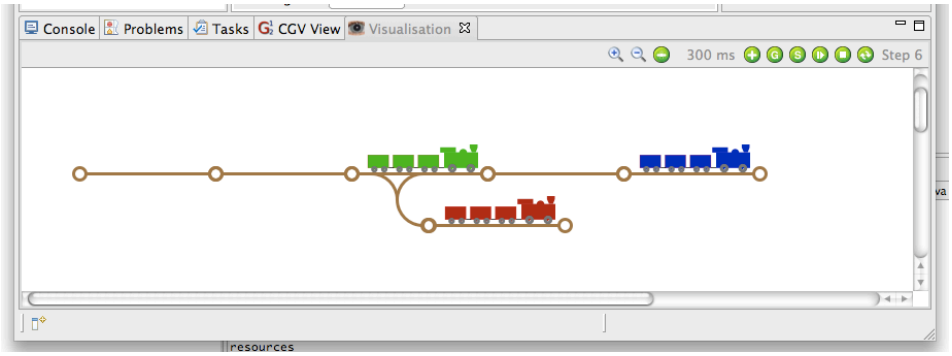

Fig. 4. 'Meet-pass problem: screenshot of a "custom" animation of the solution

figuration, for example, might be presented as a 3-dimensional picture of a car, a computer or a kitchen. Or a schedule might be represented as an animated diagram showing the workflow when the schedule is executed. The G12 toolkit includes a "custom viewer" G12-VisView, which allows the programmer to draw arbitrary pictures and to script them to depend in completely arbitrary ways on data from a file or from the solver output.

Custom views can be used during the search as well as at the end. At any stage of a backtracking search, some decision variables will have definite values, meaning that there is a partial model of the problem—generally, this cannot be extended to a complete model, since it will be amended on backtrack when the errors in it come to light. It may well be that the user-oriented picture can be partially drawn given this partial model. The variables which do not have definite values have domains of possible ones, meaning that the partial model can also be seen as standing for the set of possible states obtainable by assigning values from the domains to the remaining variables without regard to the constraints. This set of possibilities might be drawable in some way. G12-VisView can be linked to the G12 platform's search debugger so that (partial) custom views can be displayed at selected breakpoints as the search progresses. This is an important aspect of search visualation, as it shows what the reasoner is "trying to do" in humanly understandable terms.

3 Theorem proving

Theorem proving differs from constraint solving in a number of respects. Most obviously, it adresses the dual problem. Theorem proving is showing that something is necessarily the case, while constraint solving shows what is possibly the case. To derive a conclusion from assumptions, we typically add its negation to those assumptions and go through some sysyematic process of checking that the whole is unsatisfiable. To solve a set of constraints, on the other hand, we search for a demonstration of satisfiability. Other differences are only slightly less obvious: constraint solvers often search for an

optimal solution, where the concept of optimality has no correlate in the case of deduction;[3] constraint satisfaction concentrates heavily on finite domains, while theorem proving nearly always allows for infinite ones; provers are usually designed for a "thin" language of a single-sorted logic with no elaborate types, while languages like Zinc are rather rich; constraint solvers natively support numerical reasoning, while theorem provers mostly do not.

Nonetheless, some notions carry over without too much strain. The theorem proving analogue of the set of decision variables is the set of primitive function symbols of the language in which the problem is expressed. Predicate (relation) symbols count as function symbols for this purpose: they correspond to functions with value type `bool`. Function symbols of arity greater than zero are analogous to arrays, generally of unknown and possibly infinite length.

The domains of decision variables are of course the domain or domains of quantification in the intended interpretation of the proof problem. If it has no intended interpretation, then the Herbrand universe will suffice, though since this is usually infinite, some homomorphic images of it are likely to be more useful. The domain of any boolean-valued functions (predicates) is obvious.

The axioms or assumptions of the theory in which a proof is sought, including the negated goal, all go over naturally into constraints. If the theorem is provable, of course, there will in fact be no solution in the CSP sense, but this does not prevent the first order formulae, or their ground instances, from being construed as constraints on assignments.

3.1 ATP constraint graphs

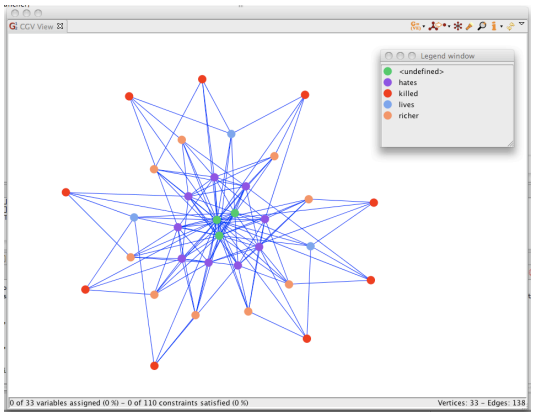

Fig. 5. 'Constraint graph of "Who Killed Aunt Agatha" with a domain of size 3

Some problems from TPTP, especially of the "essentially propositional" sort, give rise to constraint graphs which can be drawn just as in the CSP case.

[3] Proof length is not analogous to optimality of models, but rather to something like the depth, or perhaps the width, of the search tree.

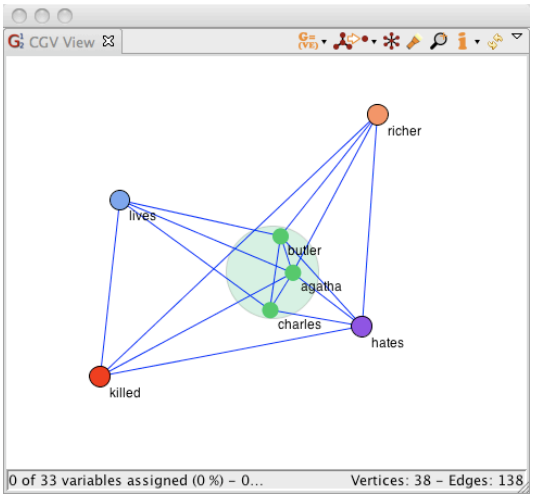

Fig. 6. Abstracted version of the constraint graph, showing the four arrays (functions) as single large nodes, except for the unclustered green one representing the three suspects

Consider TPTP-PUZ001, "Who Killed Aunt Agatha", for instance (Figure 5). For the purposes of defining constraints, the domain over which the problem is interpreted has been reduced to just three objects, but no assumption is made as to which of them is which person, or indeed whether the three people are all distinct. A greater degree of abstraction is obtained by clustering together all the vertices which represent elements of the same array (Figure 6). The cluster of non-array elements, which in this case just contains the three people, has been expanded to show the individuals. Since the arrays are shrunk to dots, this view also abstracts from the size of the domain.

Other problems, however, look less like CSPs. In many cases, there are few, if any, individual constants, and no natural domain size as the Herbrand universe is infinite. Even in such cases, however, constraint graphs can be drawn, by interpreting the language over finite domains. Although the graphs are not always informative, as they are sometimes so dense as to be little more than large cliques, there are cases in which they reveal structure which domain-specific proof searches may be able to exploit.

A related idea, little explored to date, is to use essentialy the same drawing tool to draw the unification (or resolution) graph of a first-order problem. In this graph, the vertices are clauses and the edges represent potential resolution inferences where a literal in one clause is unifiable with a complementary literal in the other. Again, in many cases this shows nothing: the graph is often very dense, consisting of little more than a big clique. Other problems have structure but it is uninteresting: the condensed detachment problems in the LCL section of TPTP, for instance, consist only of a single nucleus to which every other clause is related, so the unification graph is just a star. Some problems, however, show more interesting structure on this view, and its dynamic version may give a good animated view of the search as clauses

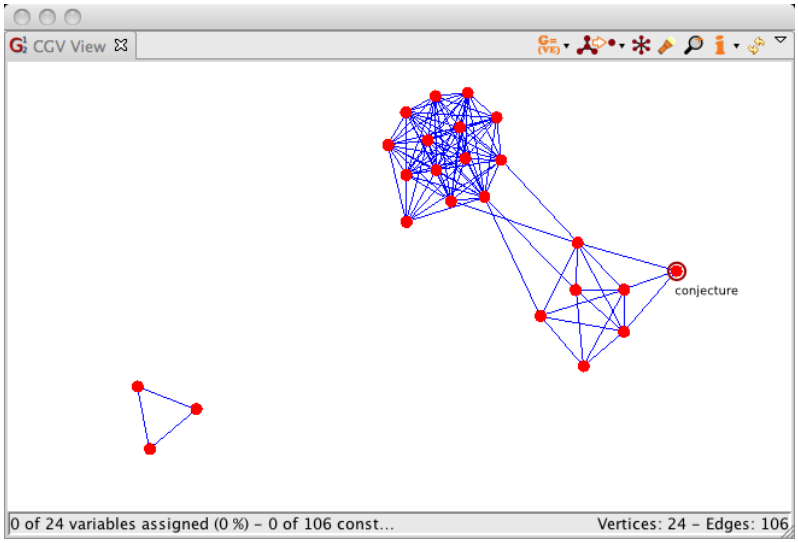

Fig. 7. Unification graph of SET013-1, with the negated conjecture ringed

are added and deleted. Figure 7 shows SET013-1, for example, pictured at the start of the search. The trio of clauses unrelated to anything else state that 'equal_elements' is an equivalence relation—a fact which is irrelevant to the proof, as they cannot resolve against anything else. The large "ball" of clauses is mostly a clique, with one more clause closely related to it, but there is other structure present as well. Since the negated conjecture is far from the clique the unification graph suggests that a strategy which causes the search to focus on the "non-clique" clauses could be very successful with this problem.

3.2 ATP search trees

The situation with regard to search trees is less happy. While a bottom-up theorem prover [4] does perform a search, this is not much like the tree-based search of a finite domain CSP solver. There are no backtracks, as the search is cumulative: the "state", considered as the set of formulae known to follow from the assumptions, increases monotonically—at least, if it is considered that the presence of a formula in it brings the implicit presence of all its substitution instances. The graph of the derivation, with the nodes labelled by formulae and the edges corresponding to inferences, is not a tree but at best a DAG. Moreover, this DAG can easily grow to have thousands of initial ("root") nodes and be extremely bushy and unstructured. If a proof is finally found, of course, only the sub-DAG ancestral to the goal (the goal is often the empty clause) needs to be reported, as this is the proof itself. The rest of the formulae derived from the assumptions are mere clutter at best.

[4] Top-down provers such as those aimed at analytic tableaux and the like are much more appropriate for the search tree viewer, but are not the focus of the present paper.

The question of how best to display the graph of derived formulae is still open. Since there are so many of them, some abstraction is needed in order to make any sort of comprehensible picture. One suggestion is to display only the number of formulae at each successive proof depth, perhaps with a little more information like the distribution of their weights, and then to allow zooming on particular formulae, showing their proofs on demand. How this might work in detail is, however, not clear. The question of what information the user or programmer is to get from this view and what it might be for would seem to be in order. For the moment, we leave it as a challenge.

3.3 ATP runtime monitoring

What is better established, and much easier to implement, is monitoring the progress of prooof search by maintaining counts of objects and events and displaying them in familiar forms such as histograms and scatterplots. The running totals of clauses generated, clauses kept, clauses used for inference, unifiers calculated, clauses subsumed and suchlike are all measures of progress and of work performed. Combined with measures such as time taken (say, in milliseconds) and memory used, they give a reasonably understandable view of the behaviour of the prover. Interesting properties of formulae, which can be plotted at the point where each formula becomes active or something of that sort, include syntactic measures like length, depth of nesting of operators and number of occurrences of some "interesting" symbol. Measures of a formula's effect on the proof search include its *fecundity*, or the number of children it has normalised by the number of potential partners with which it could combine.

Figure 8 shows examples of two ways of viewing fecundity: as a histogram or as a scatterplot. Both are, of course, very familiar styles of data presentation. The three problems shown in the figure, from group theory, nonclassical logic and set theory respectively, exhibit very different fecundity profiles. Moreover, as the scatter plots show, there are changes on this measure as the search progresses. Even in the case of GRP040, where the distribution is approximately normal, the search is not uniform but goes through phases. We do not know what these observations may mean for designing more efficient inference rules or search heuristics, but we do se them as part of learning our way about, and surely the more we look the more we shall see.

In the constraint programming case, it is common to provide some functionality for viewing search progress in the form of bar graphs and the like. Figure 9 shows how this might look when applied to first order theorem proving. The prover SOS [7] was given a problem of modest difficulty, requiring about 700 iterations of the given clause loop. At each iteration, it dumps a vector of integers to a file. These record quantities such as the runtime in milliseconds, the number of clauses generated, the number back-subsumed,

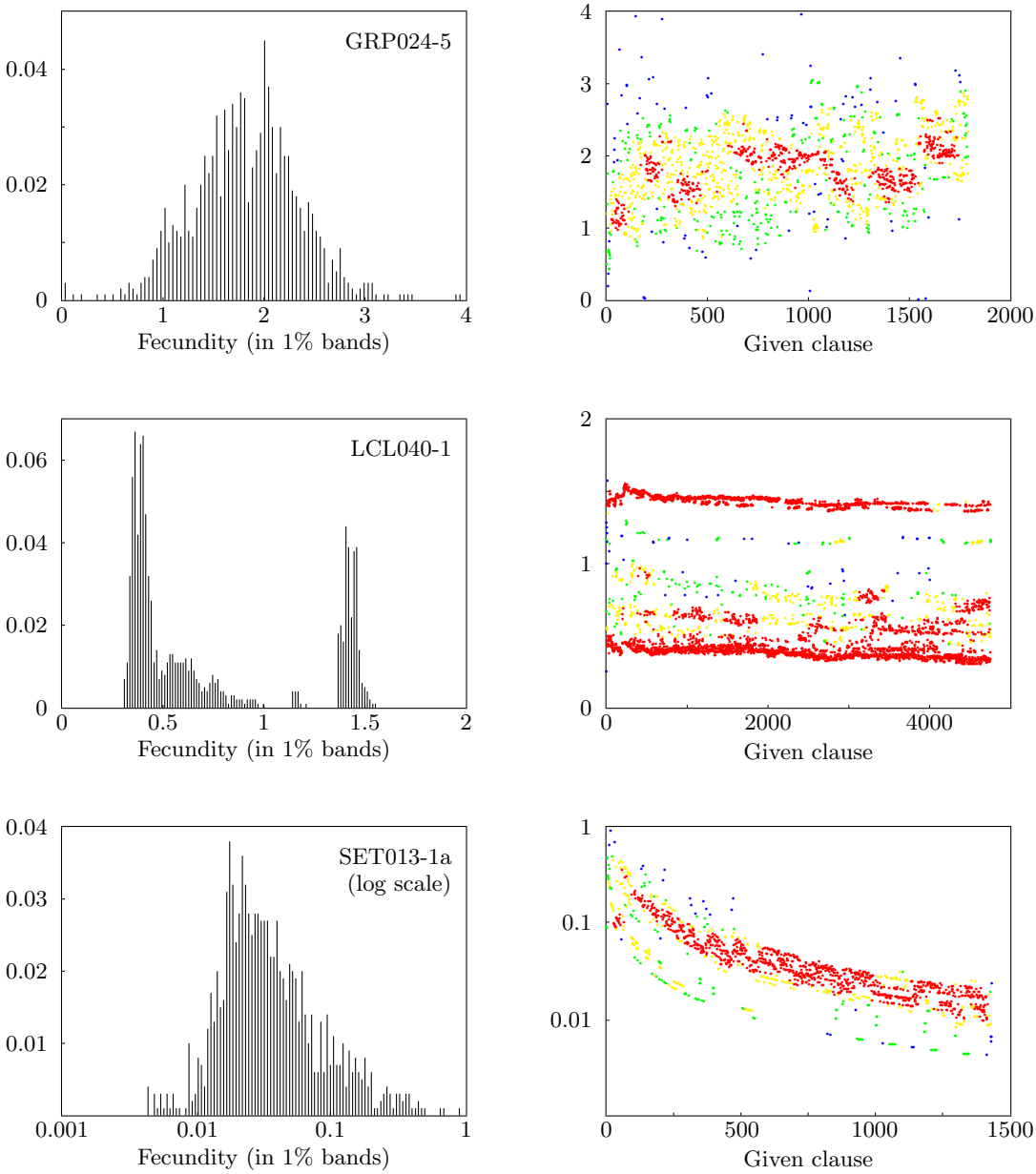

Fig. 8. Fecundity plots for three problems. Left column shows histograms of the proportion of given clauses in each percentile of fecundity. Right column shows scatterplots of the same examples: the fecundity of each given clause is represented as a dot. The colours record the density of the plot, ranging from red (dense) through yellow and green to blue (sparse).

the number remaining in the set of support, etc. A simple script computes functions of these numbers and causes them to be displayed in small windows of the type familiar from "load minitor" tools. It is easy to change the mode of each display: for instance, a number can be drawn as a spike on a bar graph or as a "needle" on a dial, or as the colour of something. Viewing several of these plots in parallel in a dashboard display can give useful clues as to how

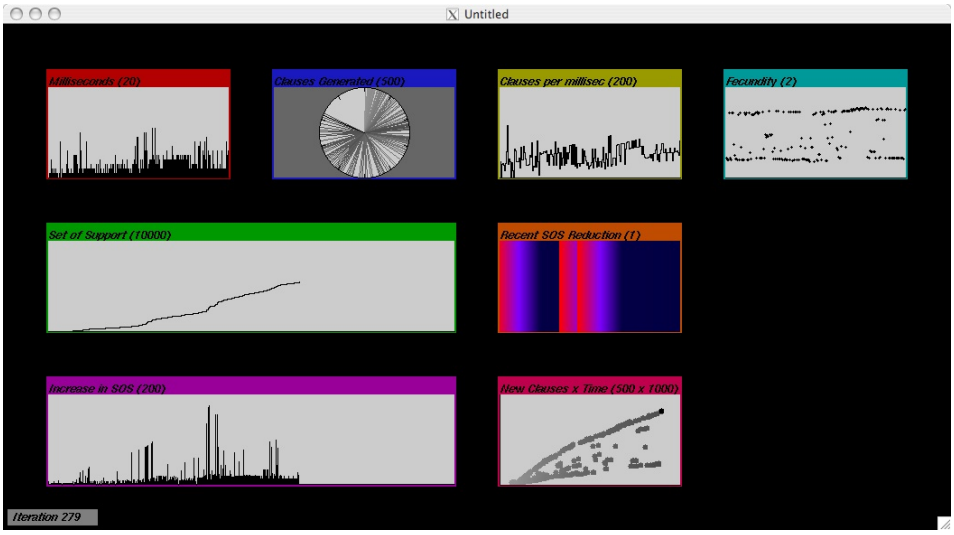

Fig. 9. Dashboard showing sliding window graphs of various quantities obtained from a runtime data dump during a proof search

things change, around points in the search where there are shifts in behaviour.

3.4 ATP custom views

Finally, there remains the visualisation option of creating a picture specific to a problem, or to a small group of problems, and animating it with data from the proof or from the proof search. This is intuitively less natural for deduction than for constraint programming, because whereas in the CP case the solution sought is a model of the clauses, describing a scenario that might easily be drawn, in the case of theorem proving the solution is a proof which shows that no model is possible.

Imagination need not be rendered inapplicable by that difficulty, however: depending on the problem, there may be ways. Figure 10 shows Aunt Agatha's tragedy again, as unfolded by Prover9. [5] This time the story is rendered graphically in terms of four figures representing Agatha, the butler, Charles and the shadowy killer who turns out to be Agatha in the dénoument. The arrows show the ground unit clauses which have been deduced by a given step of the proof search: a simple arrow where the relation is known to hold in one direction (e.g. butler hates killer), arrow with an extra "head" in the middle where it holds in one direction but definitely not in the other direction (e.g. butler hates Agatha but not conversely), double arrow where it holds in both directions, and pale crossed-out arrows for known failure. Figures are enclosed in rectangles where it is known whether they bear the relation to themselves (e.g. Agatha hates herself but did not kill herself). The snapshot shows the state just at the point where it has been deduced that Agatha is

[5] `http://www.cs.unm.edu/~mccune/prover9/`

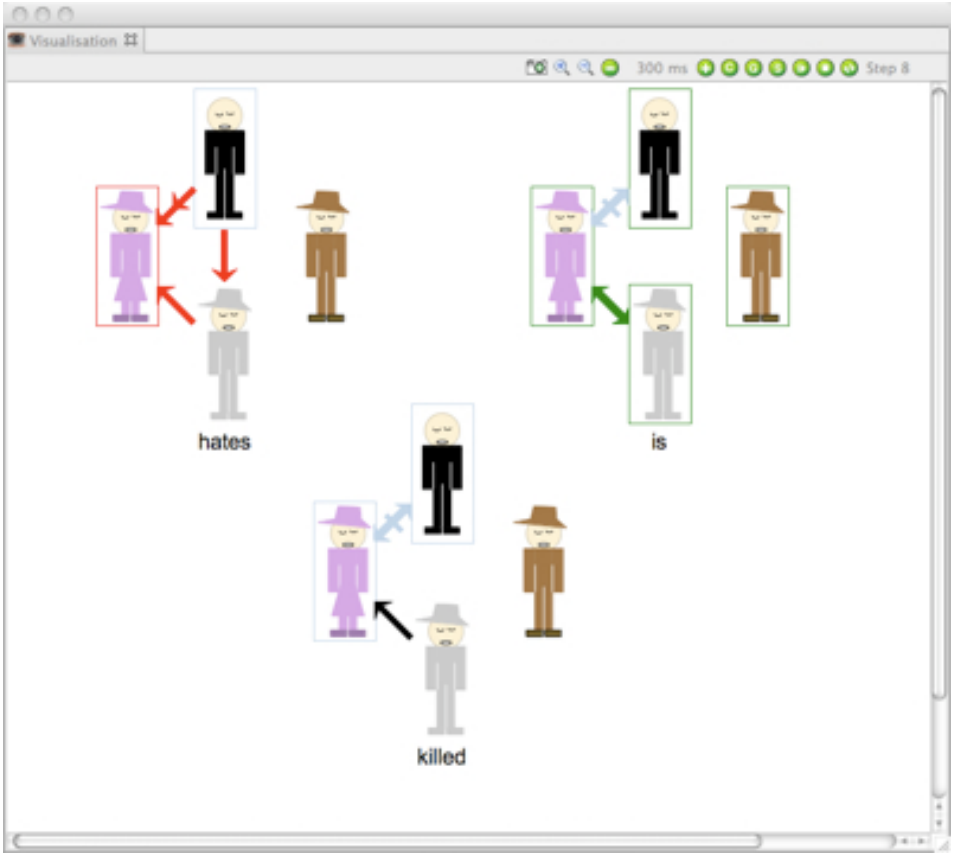

Fig. 10. An idea for what a custom view of 'Who Killed Aunt Agatha' might look like, with some conventions for showing the ground unit clauses deduced. The picture is updated every time there is new information to add to it.

the killer (green double arrow). It does give some information—for example, that nothing is known about Charles except that he is self-identical. The facts deduced to this point leave open the possibility that Charles is in fact Agatha in disguise, that Charles killed the butler, and so forth. As the sequence of images is displayed, it shows the order in which the details of the scenario are assembled, though it abstracts from any information as to which are used to deduce which others.

The Aunt Agatha story is, of course, something of an easy case for treatment in ways adapted from constraint programming, since it is a finite domain problem which is almost a CSP—indeed, it takes little work to re-express it in Zinc and have a constraint solver deal with it. The visual presentation is just a suggestion, whose utility is not really clear: nothing deep, of course, turns on the convention for using different arrows to draw a partially known graph, and there is no implied claim that the same technique will be appropriate for many other problems. It will be interesting, however, to try more examples from other theorem proving domains. We could perhaps look at some software verification problems and think about how some abstraction from the search

state could be drawn by decorating a flow chart or other such diagram.

4 Conclusion

For such purposes as performance debugging of problem encodings and partameter settings, it is essential to understand both the static semantics of logical formulae and the process of search. Traditionally in bottom-up theorem proving, the presentation of relevant information has been either too coarse (statistics on numbers of clauses generated and the like) or too fine (staring at runtime clause dumps). Visualisation tools offer the possibility of attaining better levels of abstraction. Moreover, pictures often make intuitive sense and allow important features to stand out.

In this paper, we have considered using or adapting visualilsation tools from the constraint programming community. We find that some, like constraint graphs and dashboard displays of event counts, go over quite naturally to theorem proving, while others such as search trees do not. We urge researchers in automated deduction to continue adapting visualisation tools of the kind considered here and to develop new ones. Visualisation is the most promising route to a better understanding of the automatic theorem proving process. [6]

References

[1] Choco constraint solving toolkit. http://www.emn.fr/z-info/choco-solver/.

[2] Krzysztof Apt and Mark Wallace. *Constraint logic programming using ECLiPSe*. Cambridge University Press, Cambridge, 2007.

[3] Pierre Deransart, Manuel V. Hermenegildo, and Jan Maluszynski, editors. *Analysis and Visualization Tools for Constraint Programming, Constrain Debugging (DiSCiPl project)*, volume 1870 of *Lecture Notes in Computer Science*. Springer, 2000.

[4] Christopher V. Jones. *Visualization and Optimization*. Kluwer, Boston, 1996.

[5] Nicholas Nethercote, Peter J. Stuckey, Ralph Becket, Sebastian Brand, Gregory J. Duck, and Guido Tack. MiniZinc: Towards a standard CP modelling language. In *Proceedings of the 13th International Conference on Principles and Practice of Constraint Programming*, pages 529–543, 2007.

[6] Aaron Quigley and Peter Eades. Fade: Graph drawing, clustering, and visual abstraction. In *Proceedings of the 8th International Symposium on Graph Drawing*, pages 197–210, 2001.

[7] John Slaney, Arnold Binas, and David Price. Guiding a theorem prover with soft constraints. In *Proceedings of the 16th Eureopean Conference on Artificial Intelligence*, pages 221–225, 2004.

[8] Mark Wallace. G12—towards the separation of problem modelling and problem solving. In *Proc. 6th International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR)*, volume 5547 of *LNCS*, pages 8–10. Springer, 2009.

[6] The author wishes to acknowledge the G12 IDE development team—Andreas Bauer, Viorica Botea, Matt Gray, Daniel Harabor and Andre Maroneze—for their work on the tools used in this research.