3D Visualization and Control in CLPGUI*

François Fages

Projet Contraintes, INRIA-Rocquencourt,
BP105, 78153 Le Chesnay Cedex, France,
`Francois.Fages@inria.fr`

Abstract. CLPGUI is a graphical user interface for visualizing and interacting with constraint logic programs over finite domains. In CLPGUI, the user can control the execution of a CLP program through several views of constraints, of finite domain variables and of the search tree. CLPGUI is intended to be used both for teaching purposes, and for debugging and improving complex programs of realworld scale. It is based on a client-server architecture for connecting the CLP process to a Java-based GUI process. Communication by message passing provides an open architecture which facilitates the reuse of graphical components and the porting to different constraint programming systems. Arbitrary constraints and goals can be posted incrementally from the GUI. We propose several dynamic 2D and 3D visualizations of the search tree and of the evolution of finite domain variables. We argue that the 3D representation of search trees proposed in this paper provides the most appropriate visualization of large search trees. We describe the current implementation of the annotations and of the interactive execution model in GNU-Prolog, and report some evaluation results.

1 Introduction

Several tools for visualizing the execution of constraint programs have been developed in the last few years. These tools have been found very useful for debugging and improving constraint programs, and for teaching constraint programming. One can distinguish:

- *post-mortem visualization* tools, these tools are used after execution of the program, the program is annotated with specifications of the information to trace. This approach is implemented for example in the CHIP or CIAO systems, it allows using a wide variety of viewers, including both application oriented tools [21], and generic tools, for visualizing the search tree [4, 19], finite domain variables [3], or constraint propagation [20].
- *dynamic visualization* tools, these tools are connected to the constraint programming interpreter and realize a visualization on-line, possibly with animations [12]. This approach is implemented in Grace tool [15] for finite domains visualization, and in OPL studio [2] for search tree and constraint propagation visualization.

* This communication is a revised version of [10]. The system CLPGUI is available from `http://contraintes.inria.fr/~fages/CLPGUI`.

– *dynamic visualization and control* tools which allow interaction with a CLP process through different visualizations. One example is the Oz-Explorer system [18] where it is possible to jump to any previously encountered state by simply clicking on a node of the search tree, and restart computation from that state. User-guided search is implemented in Oz-Explorer using the first-class computation spaces of Oz. Recomputation is used to trade space for time in Oz-Explorer, and similarly in OPL studio [2], the state restoration mechanisms in tree search are described in [5].

In this paper we propose to push forwards these ideas towards an open architecture for connecting a CLP process with dynamic visualization and control tools. To our knowledge, the visualization of search trees in three dimensions has not been much investigated [1]. We argue that the 3D representation of search trees proposed in this paper provides the most appropriate visualization of large search trees.

Our ambition is not to realize an *ad hoc* tool limited to a particular constraint programming system, namely GNU-Prolog [8], but a generic tool which can be ported to other constraint programming systems. The main reason for this is that a wide variety of viewers can be useful for debugging constraint programs and it is possible in this way to share developments. Our approach relies on the generic trace format which is defined in the OaDymPPac consortium [16] for post-mortem analysis. We propose to extend this format with a similar generic format for control, and to use these formats for connecting on-line the CLP process to the GUI process. Our implementation of CLPGUI in GNU-Prolog and Java is depicted in Figure 1.

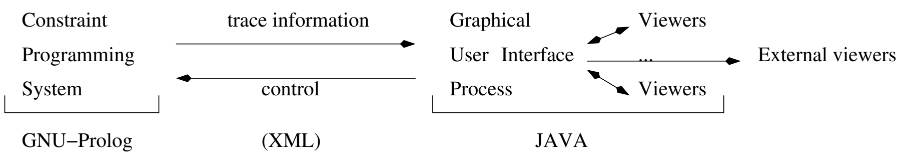

Fig. 1. Information flow for dynamic visualization in CLPGUI.

In Section 2 of this paper, we describe the client-server architecture of CLPGUI and the user console which facilitates the establishment of a connection with a CLP server, and controls the execution of the CLP program. Section 3 describes annotations that can be added to a CLP program, in order to give an external name to finite domain variables, to create buttons for posting constraints or goals from the CLPGUI console, and to define the level of granularity of the search tree which will be visualized. Section 4 presents a dynamic 3D viewer for visualizing the evolution of finite domains variables over time. Section 5 describes the representation of the explored search space by partial CSLD derivation trees, presents different visualizations with 2D and 3D viewers, and discusses their use in CLPGUI. Then in Section 6 we discuss the implementation of the interactive execution model and of the annotations, using the global variables of GNU-Prolog. Finally we provide some evaluation results, and conclude on the current limitations of the system and on some perspectives for future work.

2 Client-Server Architecture

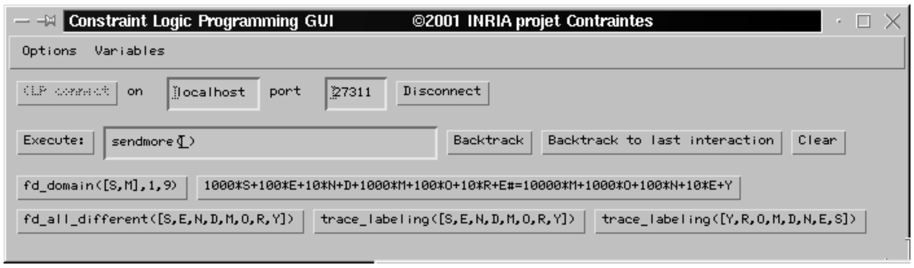

Fig. 2. CLPGUI console.

The graphical user interface of CLPGUI is a Java application connected by sockets as a client to a server which executes CLP goals. Both processes can run on different machines and communicate over the network. This has been experienced with CLPGUI for visualizing the execution of CLP programs on a Workbench of Virtual Reality. The choice of the Java language for implementing the GUI is motivated by several reasons:

- its object-orientation, all 3D viewers presented in the following sections inherit from a single class for moving and projecting 3D figures;
- the encapsulation of events handling, that is preponderant in dynamic visualization;
- the threaded execution, which is mandatory for implementing communication with the CLP process;
- its wide availability.

For efficiency reasons, we did not use the Java-3D library for the viewers presented in this paper, as they can directly benefit from *ad hoc* optimizations that speed-up their incremental display. Nevertheless the architecture can support the use of the powerful Java-3D library for developing complex application-oriented viewers.

During initialization, the CLP server starts an interpreter of the command lines received on the socket. The GUI Java client opens a graphical console such as the one in Figure 2. That console is used for establishing a connection to the CLP server, and for posting constraints or executing arbitrary CLP goals.

The CLP program may contain annotations for creating buttons for some constraints or for some Prolog goals to execute in an interactive manner. These buttons for posting constraints or Prolog goals then appear at the bottom in the CLPGUI console, see Figure 2. A click on the button posts the constraint or executes the goal associated to the button. Other arbitrary goals can be executed by entering them in a text field. In addition, one button called "backtrack" continues the execution of the current goal up to the next success, or, if there are no more success, returns to the state of the previous interaction. Another button called "backtrack interaction" forces backtracking to the state of the previous interaction. The menu bar of this console contains menus to select and activate the viewers of the search tree or of the finite domain variables.

3 Annotations

Annotations provide a simple mechanism for specifying in the CLP program the level of granularity of the data to visualize [4]. In CLPGUI, the CLP program may contain annotations for giving an external name to CLP(FD) variables, for creating buttons for posting constraints or goals from the CLPGUI console, and for marking the goals to visualize in the search tree. The following predicates are part of the annotation library:

– `varnames([V1,...,VN],[name1,...,nameN])` and `varnames([V1,...,Vn])` give an external name to a list of CLP variables. These external names are used in the graphical user interface and for the communication by sockets.
– `button(goal)` creates a button in the GUI console for executing a goal or for posting a constraint.
– `bagof_buttons(goal, call)` creates a bag of buttons for each successful instance of the second argument.
– `trace_search(goal)` executes the goal and traces the execution of that goal, by creating nodes in the search tree. The goals and constraints posted from the graphical console are always traced.
– `show_domains` updates the visualization of the current state of FD variables.

The advantages of annotations are:

– the flexibility for defining different levels of granularity concerning the information to visualize.
– the easiness for making interactive already existing programs,
– the portability of the GUI to other constraint programming systems, as all communications with the GUI are encapsulated in the implementation of annotation predicates.

The limitations of annotations are well-known in standard programming environments: they may be difficult to maintain in large programs. In that case, one solution is to automatically generate annotations with a graphical editor of the program source, where spy points and trace options can be specified. Nevertheless, one peculiarity of constraint logic programming is the conciseness of programs. CLP(FD) programs for solving combinatorial optimization problems on real-size data may compute with a huge amount of constraints and variables, but the program source for handling constraints and defining complex search strategy usually remains relatively concise. Therefore in this context, the proposed annotations appear as a satisfactory solution.

In many CLP systems however, the heuristic labeling procedures are built-in, and cannot be precisely traced with a simple annotation. In these cases, the annotations have to rely on the tracing facilities of the CLP system in order to extract, and communicate to the GUI, the relevant information. A simpler solution is to program the labeling heuristics in the host language, for making available in the host language the information coming from the constraint solvers that is relevant to the search heuristics. In that case, the effect of the search strategy can be visualized at different levels of granularity. In its simplest form, a predicate for tracing a labeling procedure can be defined with a `trace_search` annotation as follows:

```
trace_labeling([]).
trace_labeling([X|L]):- trace_search(fd_labeling(X)), trace_labeling(L).
```

It is worth noting that even if the search strategy is implemented with a meta-interpreter, or constraint posting is combined with labeling, the relevant goals of the execution can still be traced with annotations.

The same difficulty arises for tracing internal constraint propagation steps. This is not possible without access to the wakening events of the constraint solver. The annotations for tracing constraint propagation steps have thus to rely on the tracing facilities of the solver in order to extract and communicate constraint wakening events.

Example 1. The following annotated GNU-Prolog program solves the well known SEND+MORE=MONEY puzzle in an interactive manner, by creating buttons for posting the constraints and for trying two labeling goals in this example:

```
sendmore(L):-
        L=[S,E,N,D,M,O,R,Y],
        varnames(L,['S','E','N','D','M','O','R','Y']),
        fd_domain(L,0,9),
        button(fd_domain([S,M],1,9)),
        button(1000*S+100*E+10*N+D + 1000*M+100*O+10*R+E
               #= 10000*M+1000*O+100*N+10*E+Y),
        button(fd_all_different(L)),
        button(trace_labeling(L)),
        reverse(L,L2),
        button(trace_labeling(L2)).
```

This program generates the console in Figure 2. The evolution of the finite domain variables over time, after the posting of constraints and of the first labeling goal, is depicted in Figure 4. The visualization of the search tree for obtaining all solutions under the first labeling goal, and then under the second labeling goal executed after a backtracking command, is depicted in Figure 3. Under the first ordering, the labeling is deterministic. Under the second ordering, few backtracking steps occur on variable `Y` when searching for other solutions. Note that other labeling heuristics can be tried directly from the console. On such pedagogical examples, the advantage of immediately visualizing the effect of posting a constraint or trying a labeling, is clear for teaching purposes.

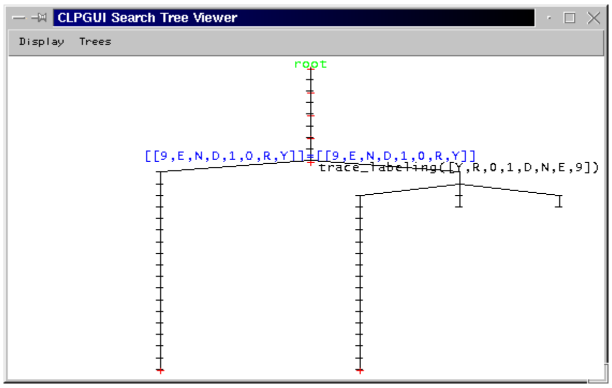

Fig. 3. Search trees with two labeling orderings in the puzzle SEND+MORE=MONEY.

Example 2. The following program solves the N queens problem by creating buttons for posting the constraints (`safe` predicate) and labeling goals for each variable (`fd_labeling` predicate) and for all variables (`trace_labeling` predicate).

```
queens(N,L):-
        length(L,N),
        varnames(L),
        fd_domain(L,1,N),
        button(safe(L)),
        bagof_buttons(fd_labeling(X),member(X,L)),
        button(trace_labeling(L)).
```

Three visualizations of the search tree for the 8 queens problem are depicted in Figures 5, 6 and 7.

4 3D Views of Finite Domain Variables

The evolution of finite domain variables is visualized in a three dimensional graph variable-domain-time, as proposed in the VIFID/TRIFID tool [22]. In CLPGUI the visualization is dynamic, the Java process reads the stream of finite domains information and paints the figure in an incremental manner. Domains are depicted by their size on the vertical axis, see Figure 4. According to options, that

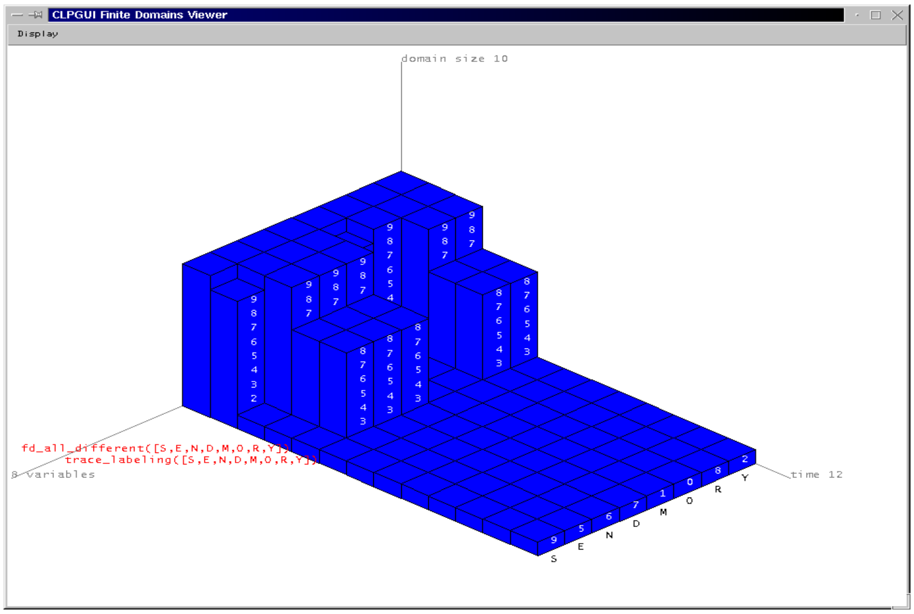

Fig. 4. Dynamic 3D view of finite domain variables in the puzzle SEND+MORE=MONEY.

can be set in the CLP program or in the GUI, only the size, the interval or the complete domain of variables is visualized. But in any case only the sizes of the domains are memorized, therefore the extra information is lost when the figure is repaint. The time axis traces the interactions (i.e. the posting of constraints in the example), and the execution of traced goals (i.e. the labeling in the example). This view shows that the posting of constraints instantiate variables S, M, O and that the first labeling step on variable `E` in fact instantiates all variables by constraint propagation. An option determines whether backtracked states are traced or erased.

The figure can be moved, zoomed and rotated. For efficiency reasons, the rotations are limited to a quadrant of a sphere which is not a real limitation for the user. In this way the visible faces are efficiently determined and the figure can be drawn incrementally.

Extra information on variables and executed goals can be obtained by moving the mouse on the position of a variable or on a time position.

The 3D dynamic view of finite domain variables evolution is very useful for teaching constraint programming. The effect of constraints is immediately seen and many strategies can be tried step by step. On large set of variables, the 3D view of domains can still be useful to get a view of the pruning power of different constraint modelings, and of the efficiency of different search heuristics, by comparing the general shape of domain reductions.

5 2D and 3D Views of the Search Tree

5.1 Partial CSLD derivation trees

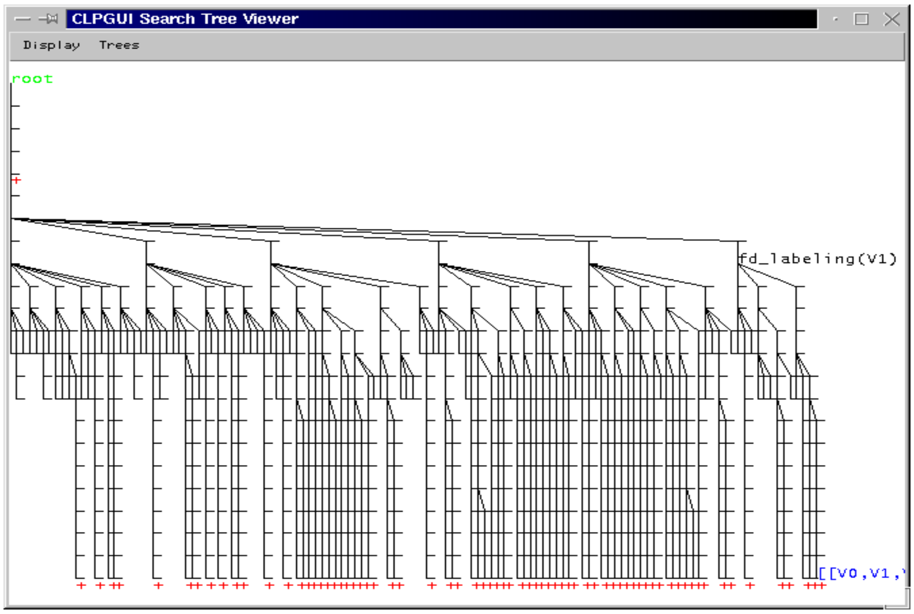

Fig. 5. 2D view of the search tree in the 8-queens problem.

The search tree considered in CLPGUI is a labeled tree defined as follows:

- a node is introduced for each call to a traced goal (called a *call node*), and for each success to a traced goal (called a *success node*),
- the label of a call node is the called goal,
- the label of a success node is the list of named variables with their value,
- the arcs correspond to the operational CLP transitions.

This tree is a subtree of the CSLD derivation tree [14]. It is thus a quite natural representation of the search tree for describing CLP program execution. A branch represents a conjunction, and the different successors of a node represents a disjunction. A success node may have several successors if there is an untraced non-deterministic goal which is executed after the success, and before the next call to a traced goal. This is the main reason why success nodes are introduced in partial CSLD trees. In this way, the non-determinism due to untraced goals cannot be confused with the non-determinism of traced goals.

One disadvantage of CSLD trees is that in the case of deterministic programs they are threadlike and thus space consuming in their standard representation.

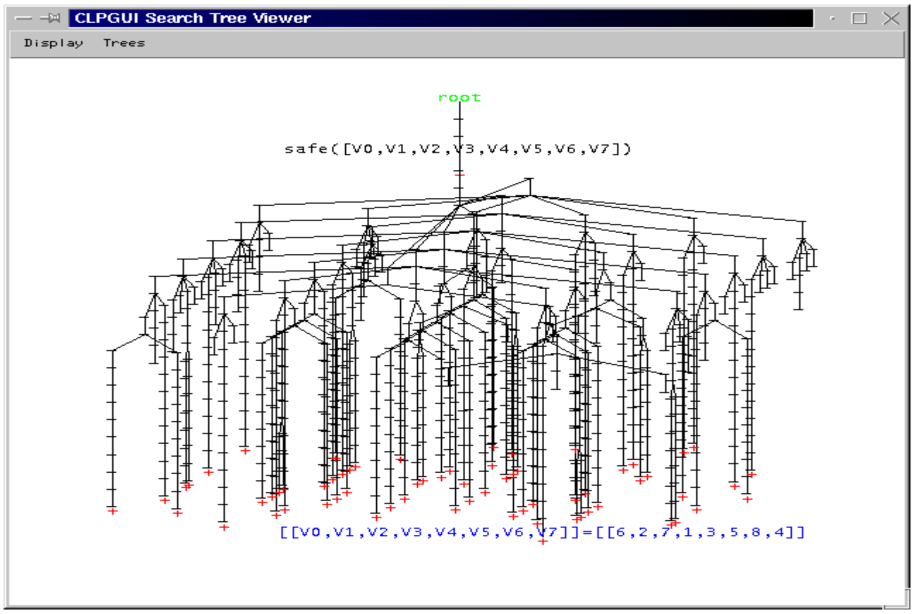

Fig. 6. 3D of the search tree in the 8-queens problem.

AND-OR trees provide a more compact representation, as the threadlike parts of the CSLD tree are compacted in the successors of a single AND-node. For this reason, in the context of logic programs where most predicates are deterministic, AND-OR trees, and their variant AORTA diagrams which indicate the status of resolution of the goals, have been preferred [9]. Nevertheless in the context of constraint logic programming over finite domains, the situation is quite different. The search tree to visualize is usually focused on the labeling predicates, or more generally on the branching procedure, which is highly non-deterministic (at least during debugging). The representation of the deterministic part of the search tree with threadlike structures provides an immediate visualization of the pruning power of constraints.

A naive solution for tracing constraint propagation steps in this approach is to add deterministic nodes for tracing constraint wakening events. For space limitation reasons, it is preferable however to aggregate constraint propagation information to the nodes of the search tree. This is proposed in the "Christmas trees" of OPL studio [2].

For search engines not based on backtracking, it is worth noting that a partial CSLD derivation tree can still provide a valid representation of the explored search space, as long as the explored states can be defined by their relation to some ancestor states. A formalization of an interactive constraint solver by transformations of CSLD derivation trees was done in [11].

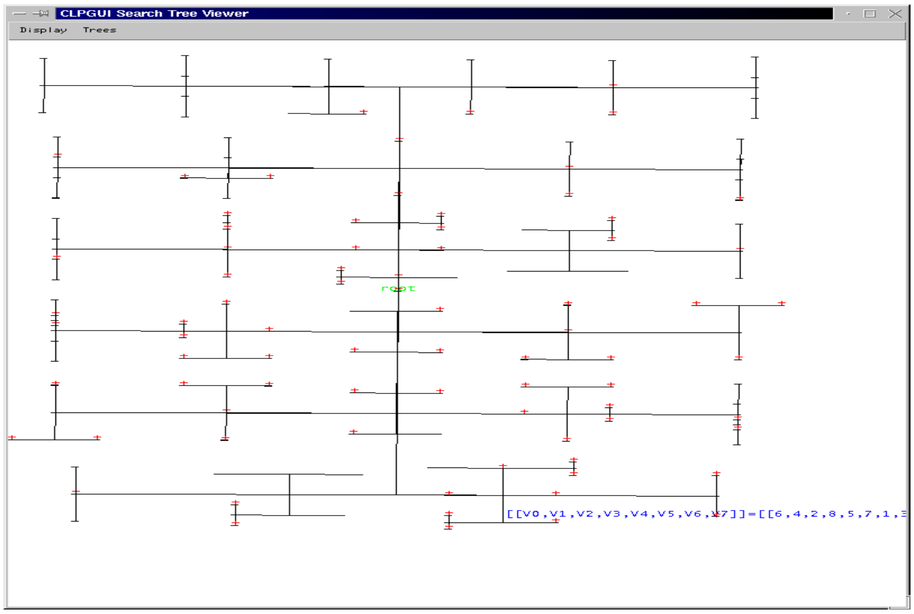

Fig. 7. Treemap representation obtained by rotation of the 3D view.

5.2 Visualization

Once the search tree is formally defined, it can still be visualized in many ways, and in some cases it can be interesting to use several visualizations at the same time. We have currently implemented several two-dimensional and three-dimensional viewers, but many more representations could be imagined and fruitfully used.

In all the following representations, the labels of the nodes are visualized when the mouse is moved on them, and an option makes all nodes visible. The successes are materialized by a red cross. Each view can be moved, zoomed and rotated.

Figure 3 uses a standard 2D representation of the search tree in a fixed width. Figure 5 uses a dynamic 2D representation of the tree with a fixed spacing between leaves. This representation of the tree can be drawn incrementally and is thus appropriate for the dynamic visualization of large trees.

To our knowledge, the 3D visualization of search trees has not been much investigated. Figure 6 shows a somewhat original 3D representation of the search tree with alternating planes of successors. One advantage of this 3D representation is that it is relatively compact, it helps visualizing rather large trees by playing with rotations, see Figure 10 for another example. Our experience is that the 3D view is the most appropriate view to apprehend the shape of large search trees.

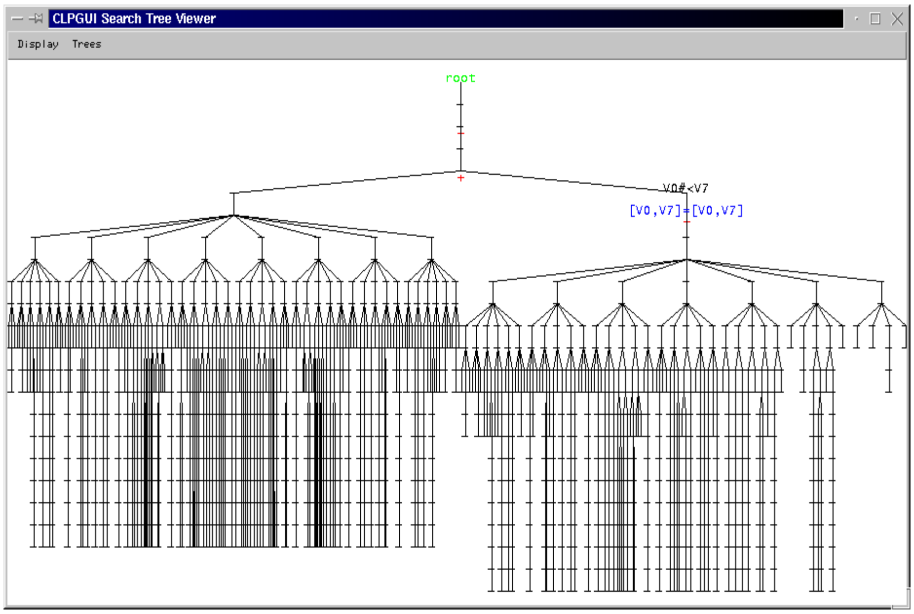

Fig. 8. Complete search trees for all solutions to the 8 queens problem, without and with symmetry breaking obtained by adding the constraint `V0#<V7`.

It is interesting to note that one obtains a treemap representation of the tree by rotation of the 3D alternate tree up to its vertical projection, as done in Figure 7. Treemap representations (with colors for aggregating information) are known to be particularly efficient to represent large data [17] and to visualize complex phenomenons such as correlations, patterns or symmetries. Figure 8 depicts the search trees for all solutions to the 8 queens problem obtained on the left without symmetry breaking, and on the right obtained by adding the constraint `V0#<V7`. Figure 9 depicts the same search trees as treemaps.

The interaction allowed in these views to restore a state is currently limited to user-guided backtracking and recomputation. The automatic recomputation of any state of the tree as described in [2, 5, 18] is currently not implemented.

5.3 Branch and bound optimization

The branch and bound procedure is widely used in constraint programming to solve optimization problems. Branch and bound optimization develops search trees in two parts. The first part corresponds to the enumeration of solutions with decreasing costs (for minimization problems). The second part exhausts the search space to show that there does not exist a better solution than the last solution found. The second part of the search tree constitutes the proof of optimality.

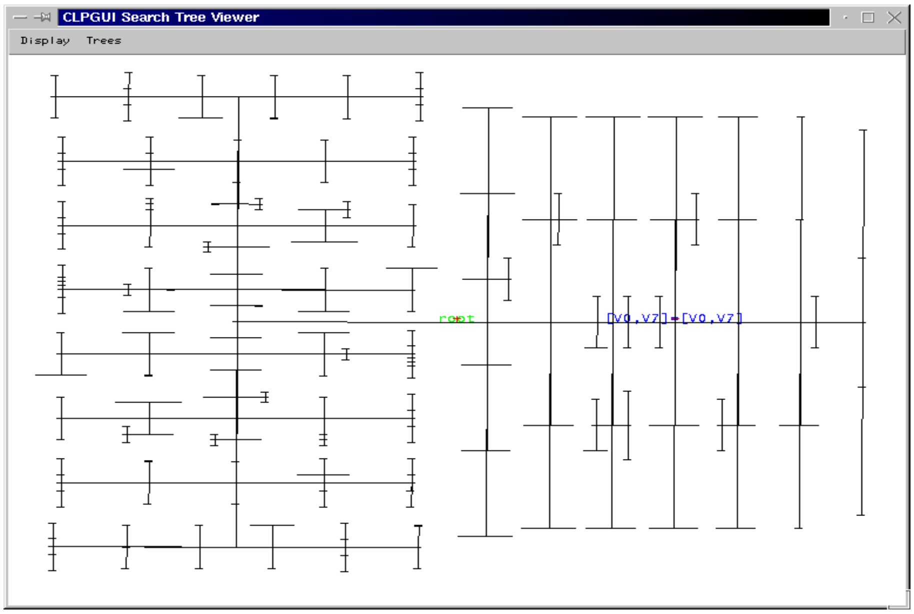

Fig. 9. Complete treemaps for all solutions to the 8 queens problem, without and with symmetry breaking.

Figure 10 shows the search tree for the bridge problem [13], a medium size job-shop scheduling problem. The first descent corresponds to the search of the first solution of cost 108. It contains 78 call nodes. The second descent (after some hesitation) corresponds to the search of the optimal solution of cost 104. It contains 99 call nodes. The bottom part of the tree contains 3728 call nodes and corresponds to the proof of optimality. The 3D view is the most appropriate view of this search tree. It can be moved and rotated without difficulty.

6 Implementation

6.1 Interactive Execution Model for CLP

The interactive execution model of the CLP process derives from a more general model for adding and removing constraints and goals described in [11]. In CLPGUI, constraints and goals can only be added to the current goal, the removing of constraints or goals occurs by backtracking. It is therefore possible on a success of the current goal:

- to add constraints or any goals to the current goal and continue resolution,
- to backtrack to the next success (command “backtrack” of Section 2),
- or to backtrack to the last interaction (command “backtrack interaction”).

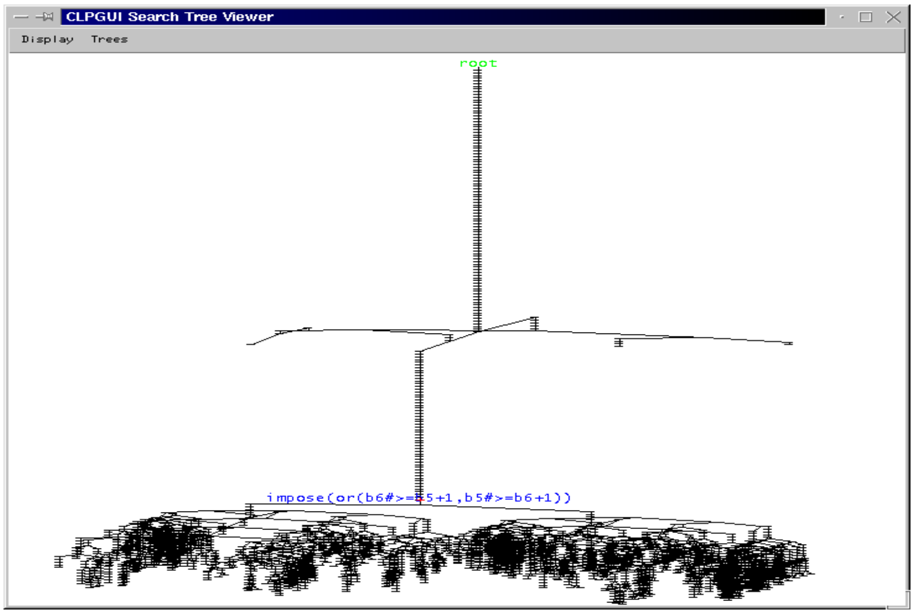

Fig. 10. 3D view of the search tree in the bridge problem (3905 call nodes).

It is worth noting that such a top level is in fact very appropriate for standard Prolog systems. Our current implementation uses the global variables of GNU-Prolog [8] to memorize global information, such as input and output sockets, variable names, and information used for backtracking. Global variables make it possible to avoid adding parameters to many predicates and lead to a simple implementation of annotations.

6.2 Communication messages

In this section we describe the communication messages which are transmitted between the CLP process and the GUI process. The CLP process produces the trace information specified by the annotations in the CLP program, or asked from the GUI. The messages emitted from the CLP to the GUI are the following:

- `<variables ...>` sends the list of FD variable names
- `<button G>` asks the GUI to create a button for posting the constraint or goal `G`
- `<undo button G>` indicates backtracking on the creation of a button for `G`
- `<node G>` traces a call to goal `G`
- `<undo node G>` traces backtracking on the call to `G`
- `<child G>` traces a success to `G`
- `<undo child G>` traces backtracking on the success to `G`

- `<undo goal G>` indicates backtracking on the call to goal `G`
- `<domainSizes ...>` sends the domain sizes of FD variables
- `<domainIntervals ...>` sends the current intervals of FD variables
- `<domainValues ...>` sends the current finite domains of FD variables
- `<undo domainValues>, <undo domainIntervals>, <undo domainSizes>` warns the GUI that the finite domain variables are updated by backtracking.
- `<success>` indicates that the current derivation is a success
- `<clear>` indicates return to top level.

In the other direction, the messages emitted from the GUI to the CLP process are the following:

- `<showSize>, <showInterval>, <showValues>` sets the information on finite domains that need be sent
- `<execute G>` asks to post constraint `G` or execute goal `G`
- `<backtrack>` asks backtracking to the next success
- `<backtrackInteraction>` forces backtracking to the last interaction
- `<clear>` asks to abort the current execution.

The portability of CLPGUI to a new constraint programming system is determined by the ability of the constraint programming system to produce and interpret these communication messages. The messages of the first list are produced by the predicates of annotation library described in Section 3. The messages in the second list are interpreted by the interactive execution model described in the previous section.

7 Evaluation

Our experience of using CLPGUI for teaching constraint programming has been very positive. The dynamic visualization of CLP programs really speeds-up the process of learning the basic concepts of domain filtering, constraint propagation and search trees. Just giving a demonstration of CLPGUI in a course provides immediate intuition to students. CLPGUI has also been fruitfully used to visualize the search tree of CLP(R) programs. Concerning debugging, CLPGUI appears as a complementary tool to standard debuggers which may present more fine grained trace information. The main advantage of CLPGUI is to immediately apprehend the shape of the search tree and the shape of domain reductions and to visualize the effects of adding constraints or of changing the labeling procedure.

On real-size data, the visualization of the search tree in CLPGUI shows satisfactory performance figures. We report in this section the timings on a Pentium III 600 MHz processor under Linux. GNU-Prolog solves the bridge problem mentioned in Section 5.3 in 100 ms, including the proof of optimality. The solving together with the visualization of the search tree with 3905 call nodes takes 470 ms with CLPGUI. This overhead is due to the communication of messages by sockets. The overhead was reduced from 2600 ms to 470 ms by optimizing socket calls. Moreover, the drawing of the tree is immediate and the figure can be moved and rotated without difficulty.

8 Conclusion and future work

We have described an open architecture for visualizing and controlling the execution of constraint logic programs. Communication by sockets between the CLP process and the GUI process has proved efficient enough for dynamic visualization and interactions. An important reduction of the overhead was obtained by optimizing socket calls and by using simple data compression techniques for communication.

CLPGUI supports the use of different viewers. Our experience has shown that a somewhat original 3D visualization of the search tree proposed in the paper, is often the preferred view to apprehend the shape of large search trees, as it is very compact. More work is needed however to parametrize the different viewers and invent novel visualizations of complex data. In this respect the flexibility of the architecture makes it possible to connect CLPGUI to external generic viewers, or to use powerful libraries like Java 3D to develop application-oriented viewers.

The most obvious limitation of our current implementation of CLPGUI is the absence of connection to a system for tracing constraint propagation, simply because such a tracer does not exist yet for GNU-Prolog. Nevertheless a generic trace format for finite domain constraint solvers has been defined in the OaDymPpac consortium [6, 16], and we plan to use this format in future versions of CLPGUI.

Finally, CLPGUI is not a visual programming tool as far as the capabilities of defining goals from the GUI are extremely rudimentary. Nevertheless the proposed architecture can support this kind of extension by adding the capability to define constraints and goals graphically, that is certainly worth investigating.

Acknowledgements. I would like to thank all members of the OaDymPpac RNTL project of the French Ministry of Research, and especially Abderrahmane Aggoun, Thomas Baudel, Pierre Deransart, Jean-Daniel Fekete, Ludovic Langevine and Mohammad Gonhiem for interesting discussions on this topic. I am grateful also to Anupam Agarwal for optimizing communication by sockets, and to Jean-Michel Leconte for his preliminary work along these lines on the workbench of virtual reality at INRIA.

References

1. P. Bouvier. Visual tools to debug prolog IV programs. In Deransart et al. [7], chapter 6.
2. C. Bracchi, C. Gefflot, and F. Paulin. Combining propagation information and search-tree visualization using OPL studio. In A. Kusalik, M. Ducassé, and G. Puebla, editors, *Proceedings of WLPE'01*, pages 27–39, Cyprus, Décembre 2001. Cyprus University.
3. M. Carro and M. Hermenegildo. Tools for constraint visualization: The vifid/trifid tool. In *[7]*, pages 253–272. Springer Verlag, 2000.
4. M. Carro and M. Hermenegildo. Tools for search tree visualization: The apt tool. In *[7]*, pages 237–252. Springer Verlag, 2000.

5. Chiu Wo Choi, M. Henz, and Ka Boon Ng. Components for state restoration in tree search. In *Proceedings of the Seventh International Conference on Principles and Practice of Constraint Programming (*CP *'01)*, Lecture Notes in Computer Science, pages 240–255, Cyprus, November 2001. Springer-Verlag.
6. P. Deransart, M. Ducassé, and L. Langevine. A generic trace model for finite domain solvers. In B. O'Sullivan, editor, *In Proceedings of International Workshop on User-Interaction in Constraint Satisfaction UICS'02*, Ithaca, USA, september 2002.
7. Pierre Deransart, Jan Małuszyǹski, and Manuel Hermenegildo, editors. *Analysis and Visualization Tools for Constraint Programming*. Springer Verlag, 2000.
8. D. Diaz. Gnu-prolog user's manual, 2001. `http://gprolog.inria.fr`.
9. M. Eisenstadt and M. Brayshaw. The transparent prolog machine: an execution model and graphical debugger for logic programming. *Journal of Logic Programming*, 5(4):277–342, 1988.
10. F. Fages. Clpgui: a generic graphical user interface for constraint logic programming over finite domains. In A. Tessier, editor, *Proceedings of the Workshop on Logic Programming Environments WLPE'02*, pages 47–61, Copenhagen, Danmark, august 2002.
11. F. Fages, J. Fowler, and T. Sola. A reactive constraint logic programming scheme. In Leon Sterling, editor, *International Conference on Logic Programming, ICLP'95*, Tokyo, 1995. MIT Press.
12. F. Goualard and F. Benhamou. Debugging Constraint Programs by Store Inspection. In Deransart et al. [7], chapter 11.
13. P. Van Hentenryck. *Constraint satisfaction in Logic Programming*. MIT Press, 1989.
14. J. Jaffar and M. Maher. Constraint logic programming: A survey. *Journal of Logic Programming*, 19,20:503–581, 1994.
15. M. Meier. Debugging constraint programs. In *Proceedings of the First International Conference on Principles and Practice of Constraint Programming (*CP *'95)*, number 976 in Lecture Notes in Computer Science, pages 204–221, Cassis, France, 1995. Springer-Verlag.
16. OADymPPaC. Tools for dynamic analysis and debugging of constraint programs. A French RNTL project. http://contraintes.inria.fr/OADymPPaC.
17. B. Schneiderman. Tree visualization with treemaps: 2d space filling approach. *ACM Transactions on Graphics TOG'92*, 11(1), January 1992.
18. C. Schulte. Oz Explorer: A Visual Constraint Programming Tool. In *Proceedings of the Fourteenth International Conference on Logic Programming (*ICLP *'97)*, pages 286–300, Leuven, juin 1997. The MIT Press.
19. H. Simonis and A. Aggoun. Search-Tree Visualisation. In Deransart et al. [7], chapter 7.
20. H. Simonis, A. Aggoun, N. Beldiceanu, and E. Bourreau. Complex constraint abstraction : Global constraint visualisation. In Deransart et al. [7], chapter 12.
21. H. Simonis, T. Cornelissen, V. Dumortier, G. Fabris, F. Nanni, and A. Tirabosco. Using constraint visualisation tool. In Deransart et al. [7], chapter 13.
22. G. Smedbäck, M. Carro, and M. Hermenegildo. Interfacing prolog and vrml and its application to constraint visualization. In *The Practical Application of Constraint Technologies and Logic programming*, pages 453–471. The Practical Application Company, 1999.