

Search-Tree Visualization

Helmut Simonis and Abder Aggoun

COSYTEC SA
4, rue Jean Rostand
F-91893 Orsay Cedex, France
email: {Helmut.Simonis, Abder.Aggoun}@cosytec.com

1 Overview

This chapter describes a visual tool for debugging and analysis of the search-trees generated by finite domain constraint programs. The tool allows to navigate in the search-tree in a flexible way and gives, for any node of the search-tree, a clear view of the current state of the program execution. The tool provides graphical representations of the form of the search-tree, of constraints and variables of the program and of the propagation steps performed after each decision in the tree. The debugger is used via a set of meta-predicates which annotate the search routine given by the user, which allows great flexibility in adapting the program to the needs of different users. The tool is now part of the CHIP constraint programming environment and covers important aspects both of correctness and performance debugging.

2 Introduction

In recent years, a significant number of applications have been developed using constraint programming (CP) technology (13) (15) (16). The complexity of problems handled is increasing and improvement of the debugging facilities becomes an urgent task. Currently, the CP technology is largely lacking debugging tools and a debugging methodology to support users. This methodology is a key point because CP programs are, different from conventional programs, data-driven computation rather than program-driven. Typical finite domain programs are structured into three parts: variable definition, constraint statement and finally the search procedure. Debugging concerns all three parts, but special emphasis lies on the search procedure, as most real-life optimization problems encountered in industry have a very large search space. To understand the effect of the search procedure on the search space there is a requirement for a novel visual tool which allows to perform an abstraction of the constraints, and which shows different views of the variables, constraints and the search space. At the same time, its use should be simple and intuitive and should not require major changes in the program under analysis. According to requirements collected from different users of CHIP and a study inside the DiSCiPI project (8), debugging tools should be useable at different levels of expertise, from a novice constraint programmer trying to understand how constraints work, to the expert programmer developing

and debugging large applications, but also by the tool developer to understand existing and to help find new or improved propagation mechanisms. It is important not only to cover the aspect of correctness debugging, finding errors in the logical meaning of the program, but also to help performance debugging, improving the speed of a correct, but slow application. The concept of global constraints introduced in CHIP (2) (4) has drastically reduced the number of constraints needed to express a problem, and allows the programmer to focus more on heuristics for the search procedure. At the same time, new debugging requirements for these powerful abstractions have arisen. This chapter discusses the search-tree visualization tool for CHIP (14) developed at COSYTEC in the DiSCiPl project. The chapter is structured as follows: We first review existing work on debugging tools for finite domain programs in section 2. We then give in section 3 a motivation why search-tree visualization is an important aspect in the development of large scale constraint applications and present the overall working principle of the search-tree tool. This is followed in section 4 by a description of the programmer's interface used. In section 5 we present the different views the tool offers. In the last section 6, we describe the current state of development and further features, which are currently under development. Visualization tools for the global constraints in CHIP are described in chapter ??, while chapter ?? gives examples and an analysis of the use of the visualization tools presented here.

3 Related Work

Given the recognized difficulty of developing correct and efficient constraint programs, there is a surprising lack of work on debugging aspects in constraint programming. In most systems, debugging tools are based on a trace of the execution. This makes it very hard to extract general information about the search procedure, and also leads to much time consuming navigation through the trace in order to find the current point of interest. The paper of Meier (11) describes GRACE, a tool to visualize domains in the context of a normal box-model trace. It can also be used to follow (in a textual form) individual propagation steps in the trace. The tool can be extensively reconfigured by the user to execute user-written code at each step of the trace process. It takes advantage of the fact that the propagation engine is written in Prolog, so that modifications can be performed in the kernel. The paper of Schulte (12), describing the OZ Explorer, is the main influence on our work. The Oz Explorer provides a graphical interface to display and control the search. It is possible to collapse or expand parts of the search-tree in order to concentrate on interesting sub-parts. A major advantage over our system is the possibility to program new search methods with a small set of primitives of the concurrent language. On the other hand, it does not contain at the moment views on constraints or on propagation steps so that its capability to follow the reasoning inside a search routine is somewhat limited. Very interesting work on visualizing search has also been done from an OR perspective (10). An early paper of Held and Karp (9) already shows

search-tree displays very similar to the format used here. In the context of the DiSCiPl project, other debugging methods known in the logic programming field, like static analysis and declarative debugging are also considered for constraint programs (5).

4 Principles of Operation

In this section we will describe the assumptions underlying our tool, the basic principle of operation and some implementation details. First we describe the different debugging problems encountered and why we have chosen the search-tree as a good candidate for debugging.

4.1 Program structure

Finite domain constraint programs typically consist of three parts, variable definition, constraint set-up and search. As the constraint solvers for finite domains are incomplete, we must choose among undecided alternatives (for example alternative values for variables) until a ground solution is found. The search procedure typically is based on depth first, chronological backtracking, but partial search methods (3) are becoming more and more popular. In any case, the search will consist of a number of nodes corresponding to states of the constraint store. Children of a node will be created by selecting an open decision and branching on possible alternatives. For the developer of an application, the most interesting aspect is the debugging of the search process, on which we will concentrate in the current chapter. Other aspects of debugging are for example the constraint set-up. If it fails, it is often quite simple to find the cause. In more complex situations, we must detect a difference between the intended model of the problem and the model described in the program. The resulting specification debugging is quite difficult and mostly a manual process. In addition to these aspects of constraint programming, there are usually program parts concerned with data gathering and preparation. These pose "conventional" debugging problems.

4.2 Symptoms

Problems in the search part will normally create one of the following symptoms:

No solution: A solution exists (for example created manually) but the search procedure fails. A good test will be to feed a known solution into the constraint solver, creating a "missing answer" problem.

Wrong solution: A solution is found but it is judged infeasible by the user. This may be caused by a bug in the application program or the constraint solver, but most often is linked to an incomplete or incorrect specification. A good test is to run the constraint program as a test only. This will often detect if the problem lies in the constraint engine.

No answer in given time: The system is backtracking in a large search space, without finding a solution and without enumerating all choices in a given time limit. It is not clear if a solution exists at all. This is a good candidate for search-tree debugging.

Answer found, but performance not satisfactory: This is the field of performance debugging. A change in the search strategy may find a solution more rapidly, redundant constraints may improve propagation, a change in the specification may make the problem harder (more propagation) or simpler (more solutions). Again, this is an area where search-tree debugging is most useful.

4.3 Operating mode

We rejected the use of an off-line tool for visualization, although these have been quite successful in visualizing logic programming languages (6). With constraint programs, too much information, i.e. all constraint propagation steps and all modifications of domains, would have to be stored, creating a very large trace file. Our tool works in co-operation with the constraint solver, replacing the usual labeling routine. In a first phase, the user defined search routine is simulated, and the structure of the search-tree stored in search node objects. These objects record the parent of the current node, the decision which is selected, and the branch which is taken. In the most basic case, the decision concerns a variable to be assigned and the choice is one particular value which is chosen. The simulation of the search procedure stops when one of the termination criteria is met:

- Number of solutions found.
- Number of failures in the search.
- Number of nodes explored in the search-tree.
- Execution time limit exceeded.

The search tree tool will stop when any of these criteria are satisfied and will display the nodes that have been explored so far, even if the user-defined search procedure did not find a solution.

Once the information about the tree is complete, the system backtracks to the state at the root of the search-tree, the tree is displayed graphically and the user can interact with the system. When a node of the tree is selected, the system will re-create the state of the computation at this node, following the path from the tree root to the node and re-enacting the decisions taken on this path. This will lead to a state of the constraint store which corresponds to the original state at this node. Navigating between nodes will cause backtracking to the root state and repeated re-enacting of the solution on the path to the selected nodes. The system must work on the full functionality provided in the CHIP environment:

- large sets of domain variables with possibly large domains.
- all basic and global constraints.
- user-defined constraints (co-routines, demons).
- predefined and user-defined heuristics.

- meta-heuristics (partial search).
- optimization meta-predicates (`min_max`, `minimize`).

In the current implementation there are restrictions on which type of constraints can be used to branch on in the search. In the following, we assume a value choice branching, i.e. at each node we choose some value for a variable.

4.4 System Requirements

The search-tree tool is conceptually quite simple, but a number of primitives must be available for an efficient implementation.

- In order to record the changing current parent in the tree, a form of trailed assignment (1) is useful. This assignment restores the previous value of the parent variable whenever some backtracking occurs.
- Both constraints and variables must have unique identifiers which can link source-level text to implementation objects.
- To follow propagation, it is required to have access to all constraints linked to a variable at all times.
- Meta-programming is very useful to simulate the user-written search procedure inside the search tool.
- In order to understand which actions are performed by propagation, we have chosen to implement *propagation events*, a log of all propagation actions (wake, update, state-change, fail) that can be accessed in an asynchronous way. We can thus access the sequence of propagation steps as data without deep changes to the propagation engine. These propagation events can also be very useful for statistics and meta-programming.

5 Interface

The interface to the visualization tool should be as simple as possible, while allowing the user to control all aspects of the display. For a novice user with small example programs it is possible to extract automatically all variables and all constraints and to display the complete search-tree generated by a standard search routine. For more complex programs, the user must annotate the source code to indicate which variables and which constraints should be displayed. The user must also annotate/rewrite custom search procedures to indicate which choices should be displayed in the search-tree. Two methods for marking variables are possible. One approach consists in marking in the source code each variable which should be handled in the visualization tool. The other approach consists in passing all variables that should be handled as arguments to a search-tree procedure. In the CHIP visualization tool we use the second alternative. The user should be able to indicate that all constraints should be included, or should be able to individually mark/unmark constraints. This can be done in the form of annotation around the constraints in the source code. By default, all constraints which use the selected variables are displayed.

For the search part, the easiest case is the use of the built-in search procedure `labeling(Terms, SelectedArgument, VariableChoice, Value_choice)`. The labeling routine performs variable selection and value assignment in a value choice scheme, where heuristics and choice functions are given by the user. In order to visualize this search scheme, the call to `labeling/4` must be replaced by a `search_labeling/4` predicate which takes the same arguments and which automates the generation of the search-tree. In addition, the `search.pl` library must be loaded.

For user defined procedures two annotations are required. One is a wrapper around the search part of the program, which indicates when the search starts and when it ends. This predicate, `search_start/2`, takes a list of domain variables as its first argument and the call to the search routine as second argument. The other predicate is a wrapper around each choice inside the search routine. The user can decide to see each choice individually or combine several choices into one. This gives additional control over the presentation of the search-tree. If several choice points are combined, the displayed depth of the tree decreases, while the width increases. The predicate `search_node/3` takes three arguments, the variable which is affected, the index of that variable in the list of variables passed as first argument to `search_start` and the call to the choice predicate. In figure 1, we show the modifications required to an application in order to use the search-tree library. The program is the well-known ship-loading problem described in (2).

```
?-lib search.

run(Start, Upper, Last):-
    data(Nr,Dur,Use),
    length(Start,Nr),
    Start :: 0..Last,
    Limit :: 0..Upper,
    End :: 0..Last,
    precedences(L),
    set_precedences(L,Start,Dur),
    cumulative(Start,Dur,Use,unused,unused,Limit,End),
    search_start(Start,min_max(labeling(Start),End)).

labeling(L):-
    search_number(L,Merge),
    labeling(Merge, 1, most_constrained, assign).

assign(t(X,N)):-
    search_node(X, N, indomain(X)).
```

Fig. 1. Programmer's Interface for Search tool

The key to success for debugging large-scale applications is the restriction of the displayed information to important parts. A large scale, industrial constraint application will often require several thousand lines of CHIP code, using a few thousand variables and several dozen global constraints, together with hundreds of simple constraints (13). The user must be able to control the volume of information both when generating the search-tree and inside the visualization tool. At each point, it is possible to reduce or to increase the amount of displayed information under user control.

6 Views

In this section we describe the different views of the search-tree visualization tool, i.e. different graphical representations of the search, the variables and the constraints of the problem.

6.1 Types of Views

The visualization tool provides a number of views into the search procedure. These views are based on four concepts:

State information is displayed for one particular state of the search-tree, e.g. the domains of all variables at a particular node of the search. Moving from one state to another updates all information.

Path information is displayed for a path in the search-tree from the root node to another node. This shows the change of constraints and variables over all choices leading to the selected node.

Tree information is displayed for all nodes below the current node. It can be shown either as the intersection or as the union of the information provided by all nodes in the sub tree. This concept is used for advanced features like propagation lifting, described at the end of the chapter.

Evolution information is displayed for all nodes in the search-tree explored before the current node. This concept is useful for statistics on failures, calls of predicates, number of propagation steps. Some of the information may be easily available for the complete search-tree, but may be quite difficult to obtain for each node in the tree without re-running the search procedure.

The diagram in figure 2 illustrates the different view concepts. We will now describe the different views and show how they relate to this general classification.

6.2 Tree View

The search process is represented in tree form with each connection from parent to child indicating a separate choice. The tree view is used to analyze and to navigate through the search space. Figure 3 shows a small part of such a search tree. It is taken from the search-tree generated by the ship loading program shown above, as are the other screen pictures used in this section.

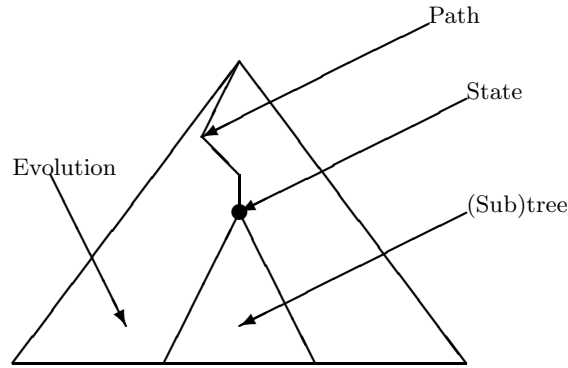


Fig. 2. View Concepts

Nodes The tree consists of different types of nodes, which are detailed below. Nodes are color coded with a user modifiable color scheme. A typical scheme would use colors to show the number of alternatives in each interior node or would show the different types of nodes in different color, as actually shown in figure 3. If a node is displayed crossed-out, then all available choices for the selected variable have been explored, i.e. there are no more alternatives to be tested. The text written inside a node can display a number of different values depending on a user selection:

- the name of the variable,
- the level in the tree,
- the index in the variable list,
- the value selected.

The root node corresponds to the state of the constraint system after the constraint set-up, before the enumeration process. At this point it is possible to see the domains of all variables after the initial constraint propagation. The interior nodes correspond to states in the search-tree where not all choice points have been explored and the constraint system has not failed.

Failure leaf nodes are displayed when the constraint system fails. Failure nodes can be shown in two ways: the more explicit representation shows each failed choice in the search tree. If for example, a indomain call tested 10 different values and each of them failed, then this representation would display 10 failure nodes. This has the advantage that there is a well-defined unique failure cause, i.e. one constraint that failed during propagation. The disadvantage is the possibly large number of failure nodes. A more compact representation shows one node for every decision predicate (like indomain) which failed. One failure node here abstracts possibly many failures, where individual values have been tested and the constraint system failed. The advantage is the more compact form of the search, the disadvantage is the difficulty to explain the propagation which led

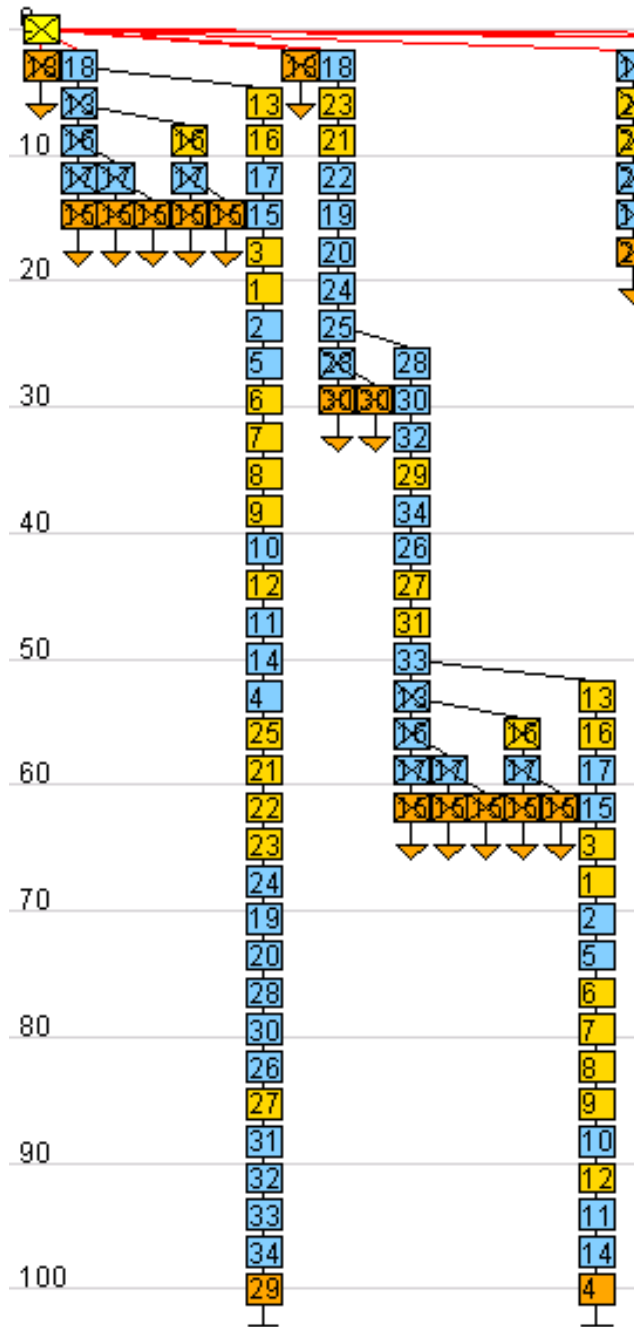


Fig. 3. Tree View

to the failure. In the current implementation only the second representation is implemented.

Success leaves are reached when all variables in the problem have been assigned and a solution was found. Note that quite often the last steps of the search procedure are induced by constraint propagation, i.e. the last real choice in the search may be several levels above the success leaf and all nodes below this choice are deterministic, i.e. the variables selected already have been assigned by constraint propagation.

In order to present the search-tree in a more compact form, it is useful to compact failed sub-trees into a single node, a failure tree. This tree is marked with the number of failure leaves it contains. The user can interact with the system to collapse or expand parts of the tree by hand. The system also has options to automatically collapse all failure trees or to expand all nodes. Similar to failure trees, the system can display several solutions as one success tree node. Again, the system indicates how many failure nodes and how many success nodes it contains. The user can collapse/expand any node to the corresponding failure or success tree.

User Interaction The user can zoom to any part of the search-tree by either using scrollbars and/or selecting an area in the current display area with the mouse. The user can navigate through the search-tree from node to node by selecting nodes with the mouse and can collapse the sub-tree originating at any node or expand a currently collapsed sub-tree. Selecting a node sets the current state of the search process to the state represented by the node. This requires to re-instantiate all variables on the path from root to the selected node to obtain the path information, which will re-run all constraint propagation on this path. The system can show information about the selected node, like the level of the node in the search-tree and the index of the selected variable.

Phase-line display There is another way to display the search tree. Instead of displaying links for the parent-child relation, we can link all nodes which assign the same variable, without changing their placement in the display. With a static variable selection order, all those nodes are at the same depth of the search-tree, so that the phase-lines are straight lines. If the variable selection is very dynamic, then the phase-lines will indicate at which level some variable is assigned. Figure 4 gives an example of the phase-line display. As we will see in chapter ??, the phase-line display is also very useful to compare the variable selection order in two different heuristics.

6.3 Variable Views

The different variable views are intended to help understand the impact of the search procedure on the different domain variables.

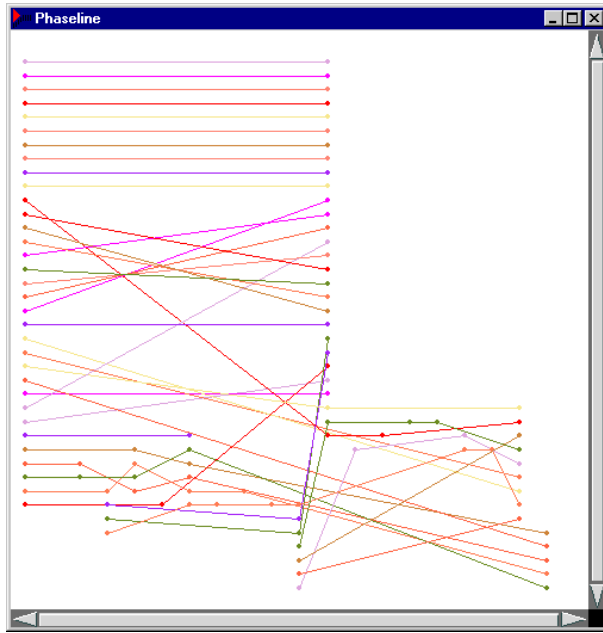


Fig. 4. Phase-line display

Update View This view shows the change of the variables on the different steps on the current path from root to selected node. In x direction, all variables are shown, in y directions from top to bottom the levels of the search-tree. Each entry in this table marks the change of the variable with different colors. A typical example is shown in figure 5, showing the update view for the first success node of the search-tree.

The following types of updates are recognized and displayed in different colors:

- variable is assigned to a fixed value in this step by search procedure (black).
- variable is ground (dark gray).
- min and max are updated (light gray).
- min is updated (light gray).
- max is updated (light gray).
- size is changed, i.e. interior value was removed (light gray).
- variable is unchanged from previous step (white).

Selecting a cell in the view will show the level and some information about the selected variable in the information display line. This view gives a good indication on how much propagation occurs in the program and which variables are influencing other variables. In the display above, we can see that only a limited amount of propagation is happening after the first variable has been assigned.

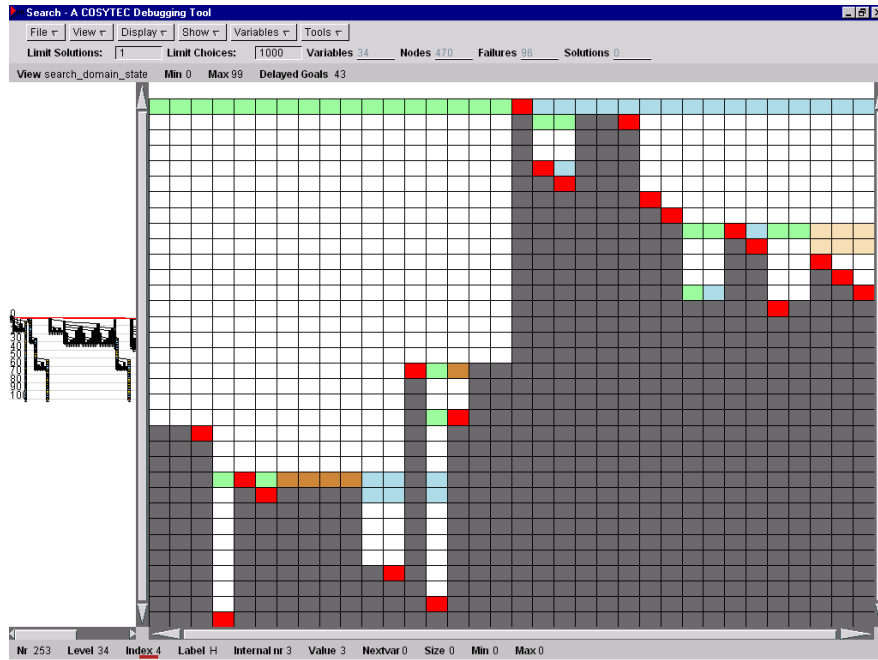


Fig. 5. Variable Update View

State View This view shows the domains of all variables at a given state. In x direction, values in the domain are shown. Each line corresponds to a variable. The entries in this view show which values are currently in the domain of the variables. They can also show which values were removed in the last assignment step. Figure 6 shows the variable state view after the first variable has been assigned.

For large domains, a textual representation of the domain may be more compact than a graphical one. It is also possible to restrict the display to a user-defined interval of values. Selecting a cell in the display will indicate which variable and which value were selected. The state view can be used to understand what information is available at a given point of the computation. We can see for example which values have been used more often than others in the current (partial) assignment.

6.4 Constraint Views

The constraint views show either the complete constraint network of the program, or show the evolution of particular constraints within the search process. The views are useful to understand the problem structure and the overall constraint reasoning. The general constraint views can be used for all types of constraints, others are specialized for the global constraints in CHIP. We only

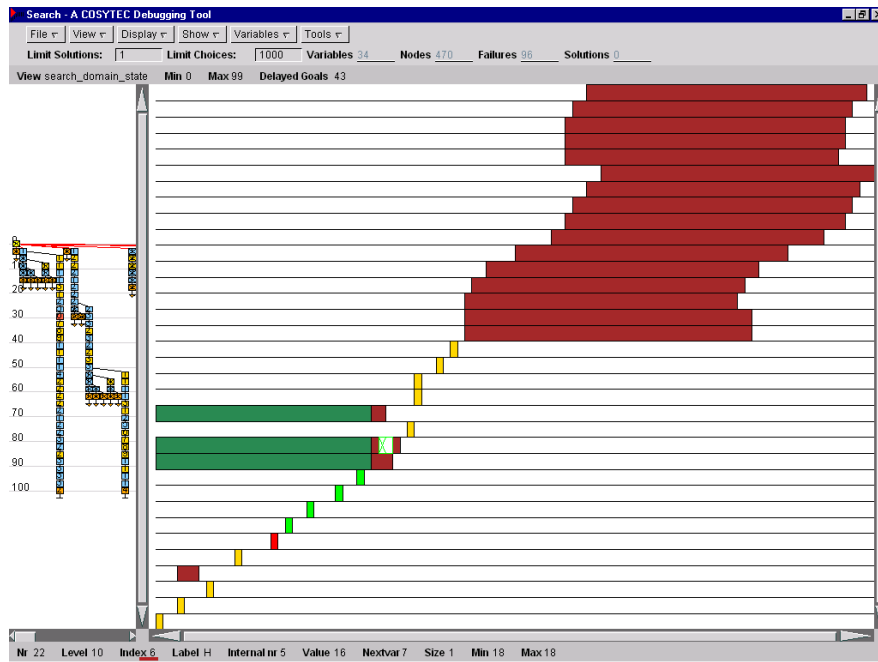


Fig. 6. Variable State View

discuss the general views in this chapter, special visualizers for global constraints are discussed in chapter ??.

Incidence Matrix All constraints can be shown as a constraint variable incidence matrix (see figure 7). In x direction, all variables are displayed, in y direction, all constraints are shown. If required, this matrix can be compressed by grouping constraints into lines which count occurrences of variables. This display gives an indication of the impact of constraints on variables, which is also used for the variable selection in certain strategies. Figure 7 shows the incidence matrix for the ship loading problem. The different types of constraints are color-coded; in this example we find a number of binary inequality constraints and at the top a single cumulative constraint which expresses a resource limit. The use of global constraints makes the incidence matrix feasible, as there are not too many constraints to be displayed. For a selected state of the search-tree, we indicate in red lines on the left and at the bottom which variables and constraints are still alive, i.e. have not been assigned (resp. solved) yet. Selecting a line in the display prints the textual representation of the constraint in the text line. The incidence matrix gives a good, compact view of the inter-relation of variables and constraints. Hidden structures and symmetries can often be recognized in this view.

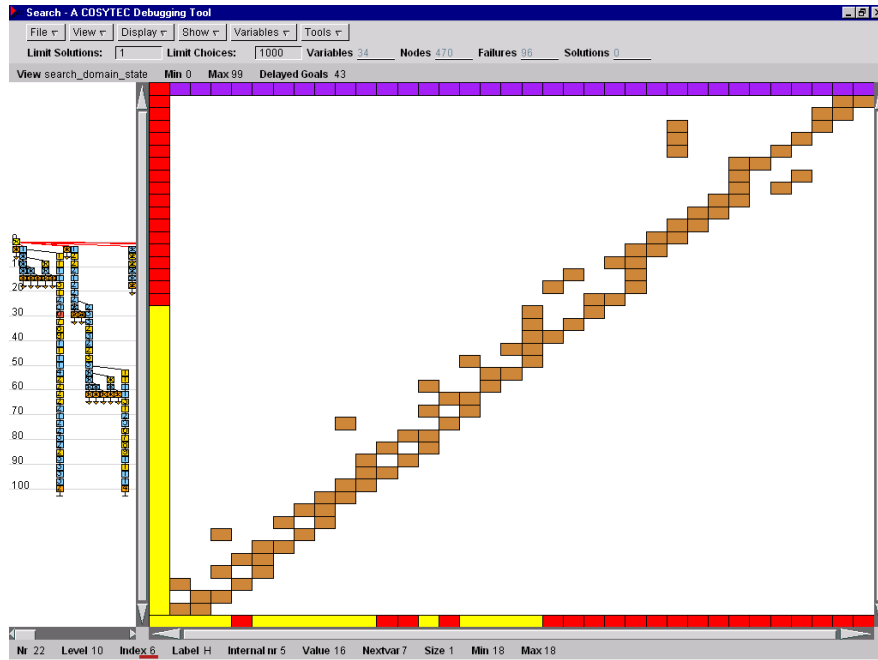


Fig. 7. Constraint Incidence Matrix

Update View This view shows the activity of the constraints on the path from root to the current node of the search-tree. In x-axis, all constraints are shown, in y-axis, the different levels of the search-tree. An entry shows what happens to a constraint at this level in the search-tree. Different colors are used to encode whether

- the constraint is woken.
- the constraint causes some domain update (min, max, remove).
- the constraint binds some variable.
- the constraint is solved.

If more than one condition applies, the stronger (lower) one is displayed. This view is useful to see which constraints contribute to the propagation, e.g. to see if some redundant constraints actually contribute to domain reductions.

Constraint Count This simple view shows a diagram with the level of the search-tree on the x-axis and the number of active constraints on the y-axis. This display shows the progress of constraint propagation as a graph.

In addition to these general constraint views, there are specific visualization tools provided for the global constraints (2) (4). Global constraints work on sets of variables using multiple propagation mechanisms for deduction. For each of these constraints, one or multiple graphical representations can be used to

display the information available to the constraint. These visualization tools are described in chapter ?? of this book.

6.5 Propagation View

The propagation views are used to understand the propagation for one assignment step. One assignment may lead (in a large problem) to several hundred constraints being woken and re-woken several times. This view is for the advanced user who tries to understand the interaction of the different constraints with each other. This display shows all propagation steps resulting from the current assignment. The view displays variables in x direction, and propagation steps in y direction from the top to the bottom. Each line of the display corresponds to one constraint. In each line, all variables which are used in the constraint are marked. Any variables which are updated are shown in a coding according to their update type. It is possible to restrict the display to show each woken constraint only once in the propagation view and to ignore all constraints which are woken, but do not further restrict variables. Selecting a line in this view will indicate the constraint and variable chosen. Figure 8 shows the propagation view of one step in the ship loading example. Setting one variable to a particular value leads to a series of updates via inequalities (each affecting two variables) and cumulative (affecting all constraints). The propagation view shows all steps of the propagation and their effect. For some choices, very little propagation will occur, some others will lead to hundreds of propagation steps. The user can zoom on the display to follow the propagation step by step. For some purposes, the view shown is not detailed enough. For global constraints in particular, we can use *propagation events* to capture the reason for individual updates of the variables. Propagation events record all waking, domain updates and failures of constraints together with information about the variable which caused the event, the variable that was affected, the details of the modification and the particular method used inside the constraint. The propagation events are written into a log, which can be retrieved later on. This information is available in the propagation-event view, which lists in a textual form all propagation events caused by some step in the search procedure. The propagation step display compresses the same information, it does not show for example the details of a domain modification.

7 Current State and Further Development

A first version of the visualization tool has been implemented and is available as part of the current CHIP 5.2 release (7). It allows the handling of small to medium sized problems (up to 500 variables). So far, neither the overhead when executing the search procedure nor the time to restate the variable bindings when selecting a node are causing performance problems. Besides extending the scale of problems that can be handled, we want to further improve the functionality. Some future extensions are listed in the following paragraphs.

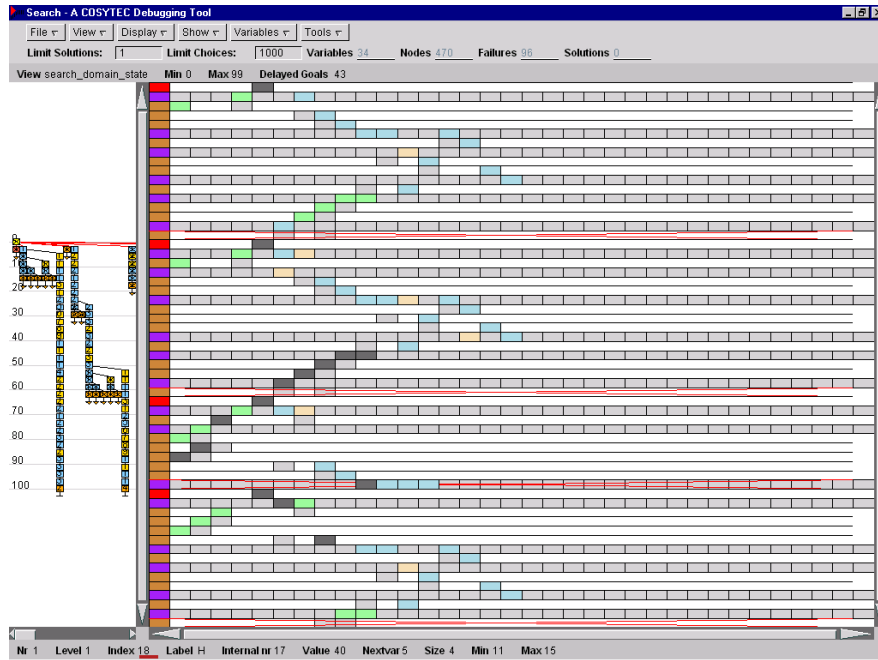


Fig. 8. Propagation View

- Improved display of isomorphic sub-trees: At the moment, a middle click on a node will find isomorphic copies of the tree under the selected node in the complete search-tree. We distinguish situations where variable and value are the same and situations where only the selected variable is the same. We want to improve this mechanism to automatically find all sub-trees for which isomorphic copies of a certain size exist. This is very helpful in understanding thrashing behavior, where the same constraint reasoning is applied multiple times.
- Why explanation for value removed from the domain of a variable: This module will find the level in the search, and the first constraint responsible for the removal of a value from a domain. While this not really gives an "explanation" for the removal, it can be used to better understand the constraint reasoning.
- Propagation lifting: If a value is removed in all leaf-nodes underneath a given node which has been completely explored, then the value can be removed at this node. Indicating such variables and values can be a powerful tool to discover missing constraint propagation.
- Failure analysis: The propagation events provide all primitives to explain failures from constraint propagation. Starting backwards from a failure, causal event links of relevant constraints can be built connecting the failure to some decision in the search-tree. Statistic analysis can also be used to understand

which constraints detect the failure with a view to performance improvement.

- Tuning of Method Selection: The global constraints use a large variety of methods to deduce new information at each step. Not all methods will be useful for all problem types. We can use the constraint views together with propagation events to determine for a given problem which methods perform most of the propagation work. This would allow to (semi-) automatically tune the engine to handle particular classes of application problems with a restricted set of methods and a corresponding gain in efficiency.

8 Conclusion

In this chapter, we have presented a visual analyzer for the search tree generated by finite domain programs in CHIP. The tool allows to navigate in the search-tree generated by the search procedure using different views for variables, constraints and propagation. The tool is aimed at all users of the CHIP system, providing different degrees of sophistication for the beginner and the experienced programmer alike. First experience with the tool has already shown its usefulness also for didactic purposes, helping to understand how constraint propagation and search work. A relatively small set of primitives is required, so that the analysis tool can be designed and modified without a deep integration with the problem solver.

9 Acknowledgment

We gratefully acknowledge the influence from many discussions with the CHIP team, in particular E. Bourreau and N. Beldiceanu, and with our partners in the DiSCiPl project, in particular the group of M. Hermenegildo at UPM.

Bibliography

- [1] A. Aggoun, N. Beldiceanu, Time Stamp Techniques for the Trailed Data in Constraint Logic Programming Systems. In Actes du Seminaire 1990 - Programmation en Logique, Tregastel, France, May 1990.
- [2] A. Aggoun, N. Beldiceanu, Extending CHIP in Order to Solve Complex Scheduling Problems, Journal of Mathematical and Computer Modelling, Vol. 17, No. 7, pages 57-73, Pergamon Press, 1993.
- [3] N. Beldiceanu, E. Bourreau, P. Chan, D. Rivreau, Partial Search Strategy in CHIP, 2nd International Conference on Meta-heuristics, Sophia-Antipolis, France, July 1997.
- [4] N. Beldiceanu, E. Contejean, Introducing Global Constraints in CHIP, Journal of Mathematical and Computer Modelling, Vol 20, No 12, pp 97-123, 1994.
- [5] F. Bueno, P. Deransart, W. Drabent, G. Ferrand, M. Hermenegildo, J. Malunzyski and G. Puebla, On the Role of Semantic Approximations in Validation and Diagnosis of Constraint Logic Programs. In Proceedings of the Third International Workshop on Automated Debugging-AADEBUG'97, Pages 155-170, Linköping, Sweden, May 1997.
- [6] M. Carro, L. Gomez, M. Hermenegildo, Some Paradigms for Visualizing Parallel Execution of Logic Programs, Proc. ICLP93, Budapest, Hungary. The MIT Press, Cambridge, MA, 1993.
- [7] COSYTEC. CHIP++ Version 5.2, Documentation Volume 6. Orsay, 1998.
- [8] M. Fabris et al., CP Debugging Needs and Tools, In Proceedings of the Third International Workshop on Automated Debugging-AADEBUG'97, Pages 103-122, Linköping, Sweden, May 1997.
- [9] M. Held, R. Karp, The Travelling Salesman Problem and Minimum Spanning Trees: Part II, Mathematical Programming 1(1971), pp 6-25.
- [10] C. V. Jones, Visualization and Optimization, Kluwer Academic Publishers, Norwell, USA, 1996.
- [11] M. Meier, Debugging Constraint Programs, In Principles and Practice of Constraint Programming, page 204-221, Cassis, France, September 1995, Springer, Lecture Notes In Computer Science 976.
- [12] C. Schulte, Oz Explorer: A Visual Constraint Programming Tool, Proceedings of the Fourteenth International Conference On Logic Programming, Leuven, Belgium, pages 286-300. The MIT Press, July 1997.
- [13] H. Simonis, Application Development with the CHIP System, Proc Contessa Workshop, Friedrichshafen, Germany, September 1995, Springer LNCS.

- [14] H. Simonis, The CHIP System and its Applications, Proc. Principles and Practice of Constraint Programming, Cassis, France, September 1995.
- [15] H. Simonis, A Problem Classification Scheme for Finite Domain Constraint Solving, Proc workshop on constraint applications, CP96, Boston, August 1996.
- [16] M. Wallace, Survey: Practical Applications of Constraint Programming, Constraints, Vol. 1, Nr1-2, pp 139-168, September 1996.