Using Constraint Visualization Tools

H. Simonis[1], T. Cornelissens[2], V. Dumortier[2], G. Fabris[3], F. Nanni[3], and A. Tirabosco[3]

[1] COSYTEC SA
4, rue Jean Rostand
F-91893 Orsay Cedex, France
email: Helmut.Simonis@cosytec.com
[2] OM Partners
Michielssendreef 42
B-2930 Brasschaat, Belgium
email: {tcornelissens, vdumortier}@ompartners.com
[3] ICON s.r.l.
Viale dell'Industria 21
Verona, Italy
email: {gfabris, nanni, tirabosco}@icon.it

1 Overview

In this chapter we describe some experiences with using constraint visualization tools developed in the DiSCiPl project for the CHIP constraint programming system. We will first discuss their use on some standard examples where we can see how constraint visualization can be used to detect performance problems and which type of improvements we can suggest. In a second part, we give an overview of some industrial CHIP applications, where the visualization tools led to significant improvements in the applications. At the end, we present an analysis of the existing tools and some directions for further improvements.

2 Introduction

The material in this chapter is derived from a tutorial on constraint visualization (14) developed at COSYTEC and from the DiSCiPl assessment report (8). The visualization tutorial is a WEB based resource, which discusses several constraint problems and their solution in CHIP. For each problem, alternative solutions are studied using the visualization tools and an explanation of the observed behavior is attempted. The assessment report gives a user's point of view on the visualization tools developed by COSYTEC and PrologIA during the project. It discusses problems of usability and of usefulness of the tools, studied on some example applications.
The debugging tools provide many different views and abstractions of the program behavior. We try here to discuss how this information can be used to improve the program. This is a first step to a methodology of performance debugging, an area recognized as "a very relevant problem" in the initial DiSCiPl

report on debugging needs (10). This chapter only studies the use of the tools, and not the tools themselves. Other chapters of this book contain detailed descriptions of the tools (see chapters **??** and **??**). We will concentrate on the use of the CHIP tools in this chapter, a discussion of PrologIA's tools (see also chapter **??**) is found in the DiSCiPl assessment report (8).

2.1 Structure of the chapter

The chapter has the following structure: In section 3, we show how the visualization tools can be used for performance debugging, improving a constraint program by adding redundant constraints or changing the search strategy. We look at five typical problems encountered in some example applications and see what information we can deduce from the visualization tools. In section 4, we look at four industrial applications that where analyzed using the visualization tools. In each case, the visualization suggested some improvements that had not been seen before. In the last section 5, we evaluate the results and suggest possible further improvements of the tools.

3 Debugging Scenarios

Developing constraint programs is still more of a craft than an engineering technique. It is quite easy to come up with an initial model which describes a set of constraints for the problem and which uses a standard search method. But that is not enough. Often, this initial model does not work fast enough, or does not find the solution (if it exists) at all. We then enter the domain of performance debugging. There are typically three approaches to improving the program:

- We can think of a new model, expressing the constraints in a different way. This may require a different encoding of the decision variables, or a new way of stating constraints. A typical example is the use of the dual model, which exchanges the roles of variables and values.
- We can add redundant constraints to the model. While these constraints are logically implied by the existing constraints, their propagation may further reduce the domains and allow us to find the solution more rapidly.
- The third and probably most common way is to change the search strategy. Instead of the standard search method, we can try out strategies and heuristics which are problem specific or which were found working for a similar problem. We might also use a partial search strategy rather than a complete one (2).

In each case, it is important to analyze the results of the programs and to compare their search spaces. Usually, this is done either by checking the execution time or by counting the backtracking steps. But these are very crude measures of performance and do not tell us much about what is really happening in each particular application. With the visualization tools, we can gain more

understanding by looking for information at a much more detailed level.
In this section we try to cover typical situations which we have encountered in many applications. We will describe five scenarios, each concentrating on one aspect of performance debugging:

- finding new variable/value orderings
- comparing two heuristics
- adding redundant constraints
- discovering structure in the problem
- identifying weak propagation

Note that in large-scale problems these issues do not arise independently, but must be treated together at the same time.

As examples we use classical CHIP demo problems, the N-queens problem (15), a map coloring problem (11) (15), the ship loading example (12) (1) and the square placement problem (1) (4). All of them have been solved before the visualization tools were available. But in each case, we have found new improvements when we re-considered the programs with our new tools. For space reasons, we can here only concentrate on some aspects of these improvements. A more detailed description of the problems and the different constraint models is found in (14).

3.1 Finding new variable/value orderings

When we develop a constraint application, the constraint model always is particular to the problem at hand, since normally the constraints are directly derived from the problem. For the search routine, on the other hand, we often start with a standard built-in strategy. A typical example would be to use the built-in `labeling/4` routine which either takes the variables in the input order or selects them according to a pre-defined selection strategy like *first fail*, and which uses the standard `indomain/1` enumeration trying the values in the domain in increasing order. Quite often, this type of strategy will lead to a solution (after some search perhaps), but for some problems, it may not work at all. We will now see how we can use the search-tree tool (see chapter **??**) to find new variable and value selection strategies. These strategies will be more adapted to the particular problem we want to solve and can take some domain knowledge into account.
As an example, we consider the classical N-queens problem with a standard labeling routine of *first fail* variable selection and an `indomain` value selection. The CHIP program for this problem is shown below with the annotation required for the visualization tools:

```
?-lib search.
?-lib visualize.

top:-
```

```
        solve(40).

solve(N):-
        length(L, N),
        L :: 1..N,
        create_dif(L, 1, N, L1, L2, K),
        alldifferent(L),
        alldifferent(L1),
        alldifferent(L2),
        visualize(L,visualize_variable,
                  [winw<-500,winh<-500],queen1),
        search_start(L, label(K), [winw<-760,winh<-500]).

create_dif([], N, M, [], [], []).
create_dif([H|T], N, M, [H+N|R], [H+M|S], [t(H,N)|V]):-
        N1 is N+1,
        M1 is M-1,
        create_dif(T, N1, M1, R, S, V).

label([]).
label([H|T]) :-
        delete(Var, [H|T], Rest, 1, first_fail),
        Var=t(X,N),
        search_node(X,N,indomain(X)),
        label(Rest).
```

If we test this program for increasing values of N, we can observe the following behavior. In figure 1 we plot the number of backtracking steps needed to find a solution over the increasing board size from 4 to 200. The output is obtained by one of the visualizer tools for CHIP, the `visualize_measurement` tool which displays the number of backtracking steps. We stop the search if we do not find a solution within 5000 backtracking steps. For small sizes of N, we find solutions quite rapidly, with some exceptions which need up to 2000 backtracking steps. But for larger values of N, we quite often do not find a solution within the given limit. Clearly, this default search method is not appropriate for large problem instances.

To analyze the problem more closely, we look at the search-tree generated for problem size 40 and select the node of the first failure (figure 2).

We see that failure occurs rather late, when nearly 90 % of all variables have been assigned. We can also see that we only backtrack for a few levels before we correct the problem and find a solution. In the domain state view on the right we see that most variables are already assigned to values. Each line of the display represents one variable, fixed values are shown in light gray with the currently assigned variable shown in black, the domains of the remaining variables are shown in darkgray and the constraint propagation shows crossed out all values which are removed in this assignment step. Note that some color transformation

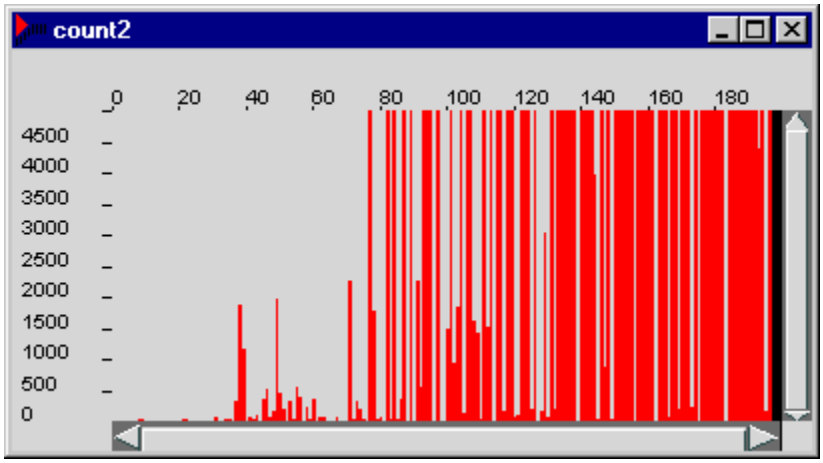

Fig. 1. Counting backtracking steps for first fail

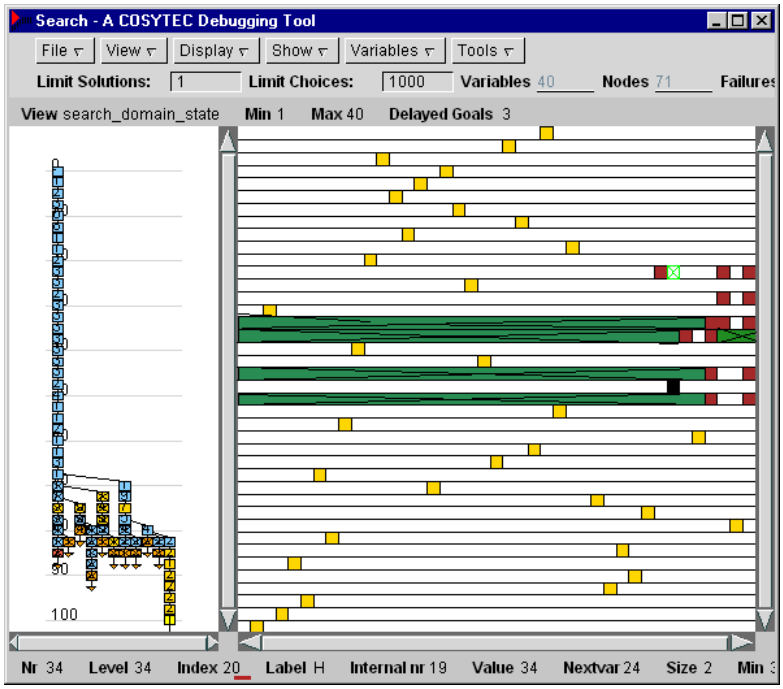

Fig. 2. 40-Queens problem, domain state at first failure

was performed on the pictures to make the information appear more clearly in black and white. Looking at this display, we can make three observations:

- All remaining unassigned variables are in the middle of the board. The *first fail* strategy did select the variables in a dynamic order. The diagram in figure 3 shows the variable selection order in more detail.
- All remaining unassigned values are at the top of the domain. The smaller values have been used up, so that only the larger values remain. This is not surprising, the `indomain` value choice will always try the small values before selecting a large value.
- If we count the number of remaining variables and the number of remaining values, we note that there are 6 remaining variables, but only 5 remaining values. Obviously, there can not be any solution at this point, because all variables must have pairwise different values. The forward checking of the `alldifferent` constraint does not detect this. We will come back to this observation later on.

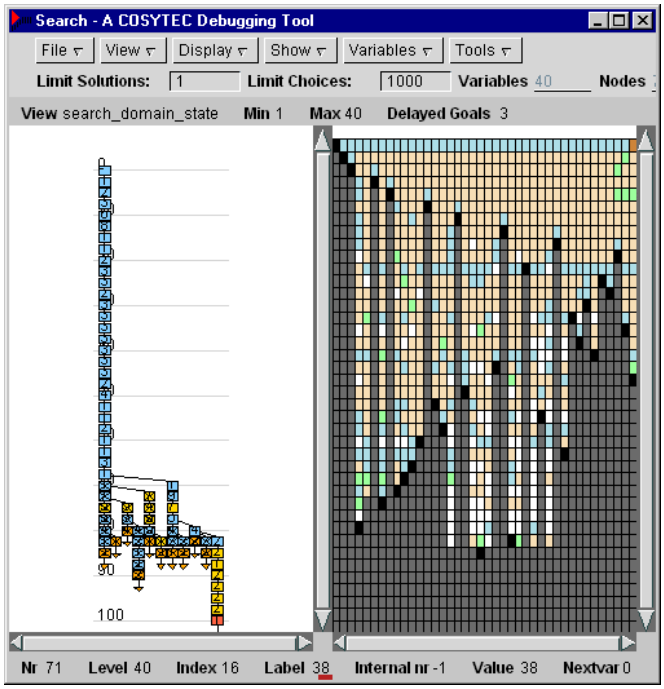

Fig. 3. 40-queens, variable selection

In figure 3 we see the domain update view. Each line is one assignment step, each column is a variable. The variables in black are assigned at this step, the

variables in dark gray are already fixed. The lighter colors show different types of domain updates. We can see that initially variables on the left of the board are selected, but that the strategy starts picking variables dynamically and that the last variables assigned are in the middle. In the picture we see the path to the first solution, and we can note that when the 7th last variable is chosen, all remaining variables are assigned by propagation.

If we look at the tree for other problem sizes, we observe the same behavior. The remaining variables are in the center of the board and the last unassigned values are the high values. This suggests an assignment strategy which tries to change this. If we start assigning the variables in the middle first, then some other variables, perhaps easier to assign, will be left at the end. But as a dynamic variable choice is usually better than a static one, we keep the *first fail* strategy, and just reorganize the variables in a different initial order. For value assignment, we could try and use `indomain(X, max)` which tries to assign values starting from the largest value. But, while this would avoid the problem observed, it would just mean that in the end we are left with unused small values instead. A better choice will be to use `indomain(X, middle)` which starts assigning values in increasing distance from the center of the domain. This values choice does not have a bias for either small or large values (another idea would be to use a randomized value selection). Implementing this new strategy, we find the following search-tree (figure 4):

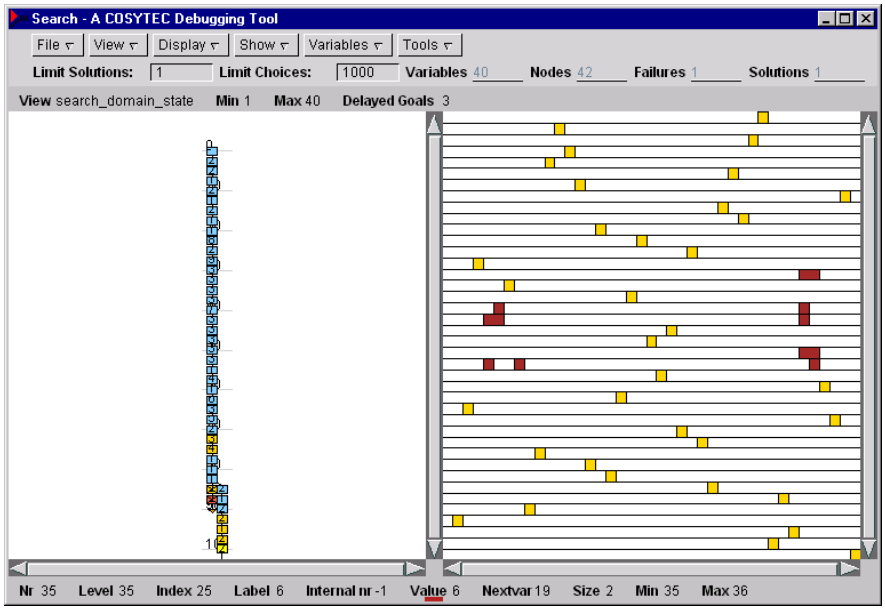

Fig. 4. 40-queens, center labeling, first failure

We can see that the search-tree is smaller for this problem, there is only one backtracking step. Selecting the point of the first failure, the last remaining values are both large and small values, all values in the middle have been taken. The last unassigned variables are still in the middle of the board, even though we started the assignment there. But clearly, one problem instance is not enough to decide if this new method is an improvement or not. If we again run all problem instances from 4 to 200, we obtain figure 5.

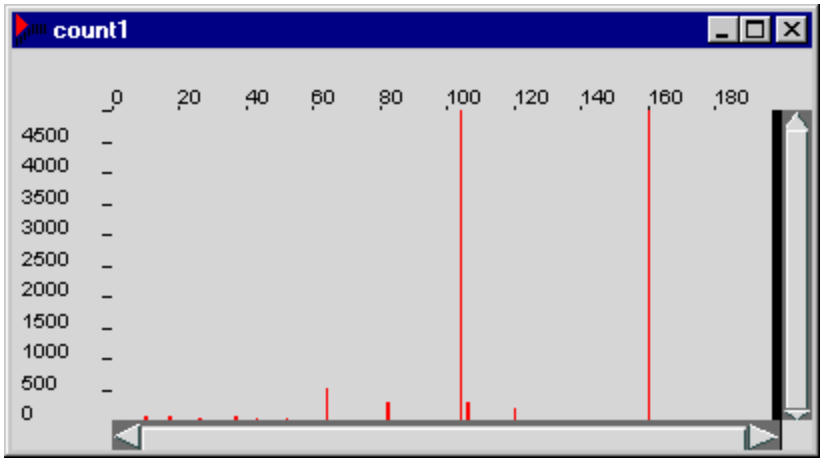

Fig. 5. center labeling, all problem instances 4 to 200

Clearly, this is a much improved search strategy. There are only two problem instances where the system does not find the solution within 5000 backtracking steps, and usually it requires a lot less choices than before. If we want to refine the solution to avoid the two exceptional problem instances, we have to introduce partial search techniques, *credit based search* (2) works very well for this problem.
How general is this technique to deduce heuristics from the search-tree and domain information? In (13), we described a similar analysis for the Mystery shopper problem (9), where a different value choice strategy suggested by a failure analysis leads to a no-backtrack solution for CHIP, while with the standard search strategy no solution could be found.

3.2 Comparing two heuristics

Quite often, in developing search strategies we can come up with two rather similar variants of a program. It is interesting to see the difference between these variants, not only in the final solution, but also in the deduction steps which are used. As an example we can compare two variants of the *first fail* heuristic for the N-queens problem. The first variant uses the `delete/5` primitive in a recursive

user-defined search procedure, the other uses the `newdelete/5` primitive. Both functions select from a list of variables the first variable which has the smallest domain and also return the remaining elements of the list. The difference lies in the way the list of the remaining elements is constructed. `newdelete` returns the list in input order, while `delete` reorders the list, which is somewhat faster. This means that at some later point in the search `delete` may pick a different candidate than `newdelete`. This slight variation can have a significant impact on the search. To visualize the difference, we generate a search-tree which shows both strategies next to each other. This can be done with the code shown below.

```
compare(K):-
        once(label(K)),
        fail.
compare(K):-
        once(label1(K)),
        fail.

label([]).
label([H|T]) :-
        delete(Var, [H|T], Rest, 1, first_fail),
        Var=t(X,N),
        search_node(X,N,indomain(X)),
        label(Rest).

label1([]).
label1([H|T]) :-
        newdelete(Var, [H|T], Rest, 1, first_fail),
        Var=t(X,N),
        search_node(X,N,indomain(X)),
        label1(Rest).
```

Figure 6 shows the resulting trees for the 40 queens problem, on the left is the selection with `delete`, on the right the selection with `newdelete`.

Superficially, the trees look quite similar, but checking the selected variable at each step shows some difference rather early on. This difference becomes more clear with a *phase-line* display (figure 7), which links all nodes in the tree which select the same variable. This views indicates when some variable is assigned in different parts of the search tree.

Figure 7 shows that initially, the same variables are selected in both trees, but that then the order of selection differs significantly. We can also see that even for one strategy the selection is not always the same, on backtracking other variables are more constrained and therefore selected first. The diagrams above have shown that the search-tree and the selection for the two very similar search strategies vary even for the quite simple N-queens problem. But perhaps this is just a matter of performing the same steps in a different order, and the resulting solutions are quite similar? Figure 8 below demonstrates that this is not the case.

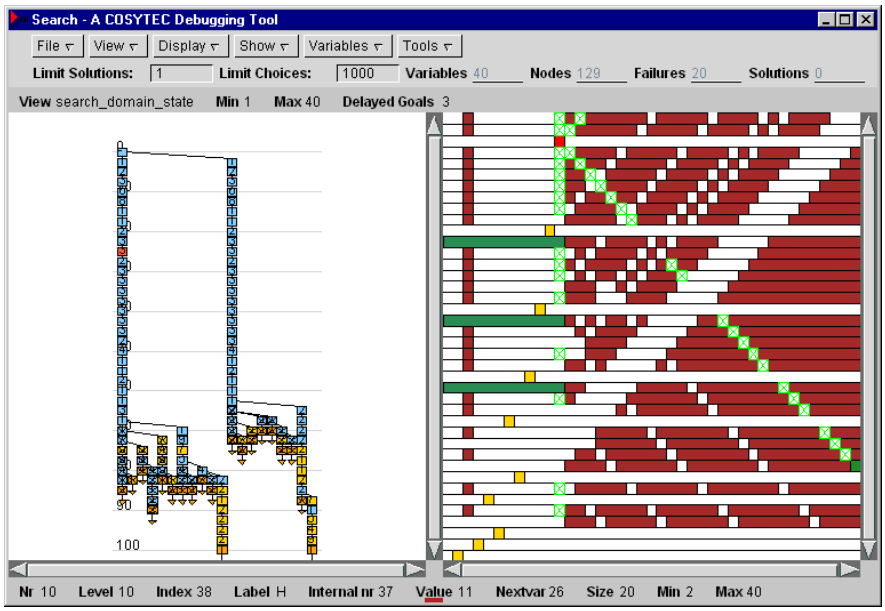

Fig. 6. Comparing strategies

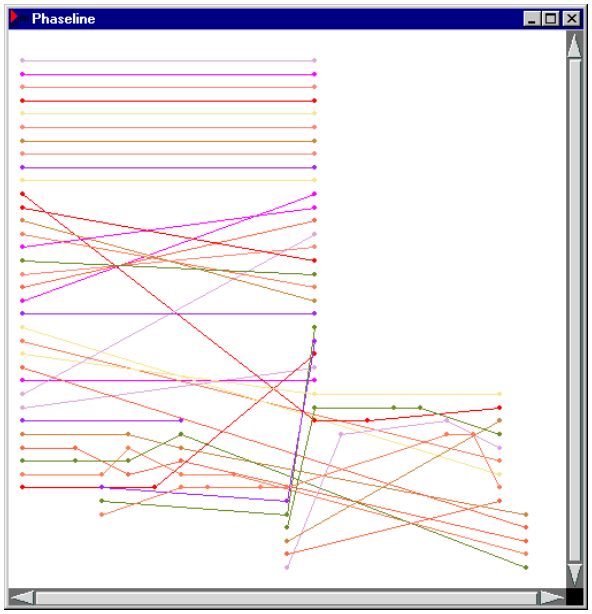

Fig. 7. Phase line display

The figure shows an overlay of the two first solutions that are found. Queens in black are assigned in the common part of the search-tree, the gray colored queens show the assignment where the two solutions differ, there are only two queens outlined in black which are placed in both solutions to the same location after the search-trees diverge. The small difference in the selection function leads to two quite different solutions. This technique of comparing two search-trees can be applied in many other situations, for example to check the impact of some redundant constraint, or the effect of some small change in the input data. The *phase-line* display allows to identify quickly if there are major differences in the variable order, it is also possible to identify *isomorphic sub-trees* (see chapter **??**).

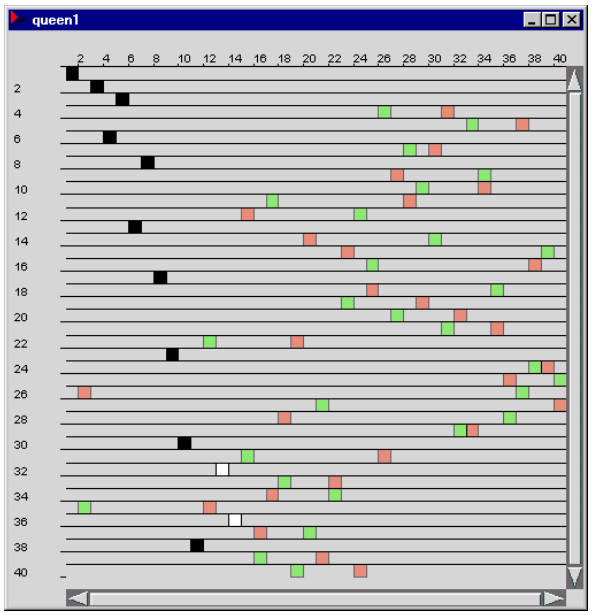

Fig. 8. Overlay of solutions

3.3 Adding redundant constraints

In many constraint programs, the results can be significantly improved by adding redundant constraints, i.e. constraints which are logically implied by the original constraints, and which are only added to improve propagation. Obviously, this improved propagation must be balanced against the cost of processing the new constraints. To understand the value of a particular redundant constraint, it is useful to see the effect on the search-tree and on the variable domains. As an example, we choose the ship loading problem from (12), which is used for one of the standard CHIP examples (1). This problem is a simple resource restricted

scheduling problem, where tasks, linked by inequality constraints, are also requiring different amount of manpower resources, with an overall limit on the available manpower at each time point. This problem is modeled with inequality constraints and a `cumulative` (1) constraint in CHIP. But there is another possible, redundant constraint that can be used, the `precedence` constraint (3). In figures 9 and 10, we show the search-tree without and with the `precedence` constraint. Note that the tree without the redundant constraint has 470 nodes, and the tree with the redundant constraint only 287 nodes. But due to the dynamic variable selection, the trees are not directly comparable, as the variables are assigned in a different order.

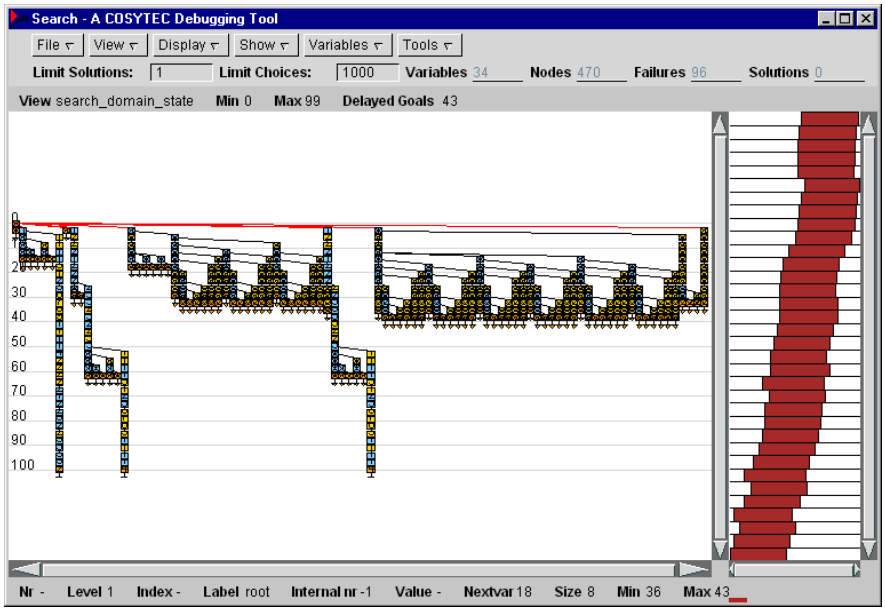

Fig. 9. search-tree without redundant constraint

We can also compare the domains of all variables after the constraint set-up, before the search starts. The first image (figure 11) shows the domains without the added redundant constraint, the second one (figure 12) shows the domains with the added redundant constraint. It is clear that the domains are much reduced.

Our visualization tools currently do not provide any way to compare run times of different strategies or constraints. Due to the instrumentation of the solver generating the tree, this would not show the real execution times in any case. To measure execution times, it is still required to run both programs with the same data-sets and compare overall results. The search-tree and visualization tools at the moment only allow us to get a rough idea of the value of redundant

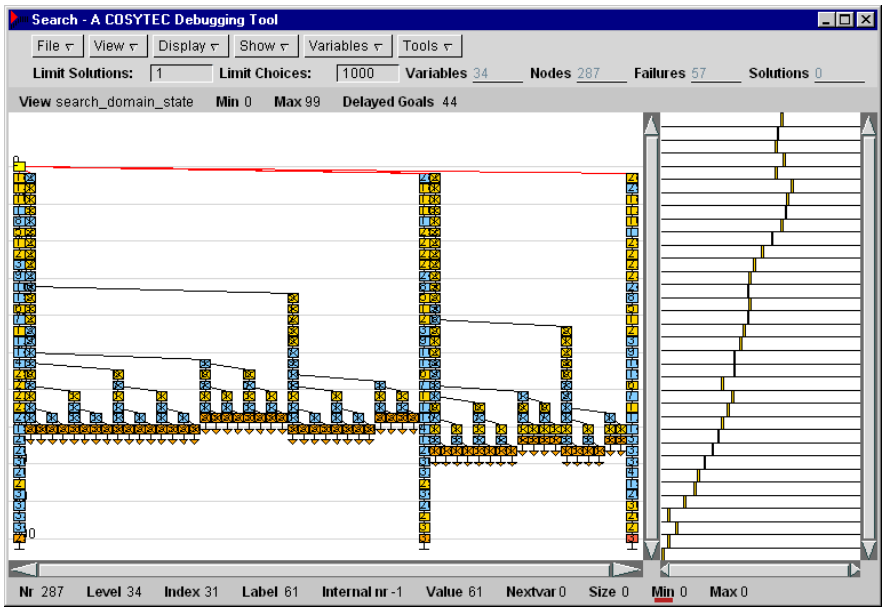

Fig. 10. search-tree with redundant constraint

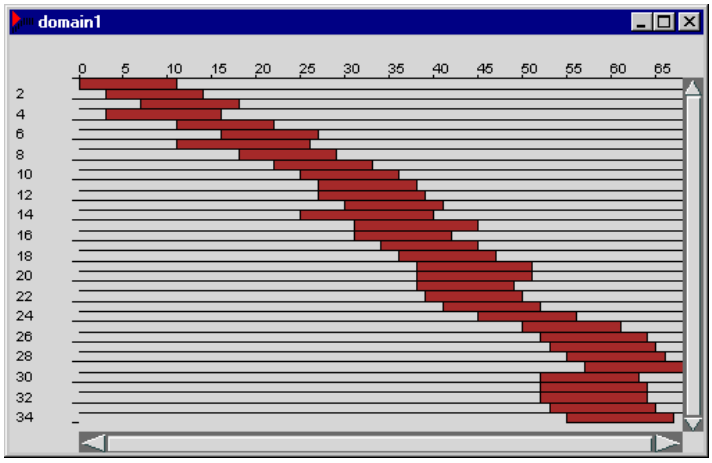

Fig. 11. Initial domains without redundant constraint

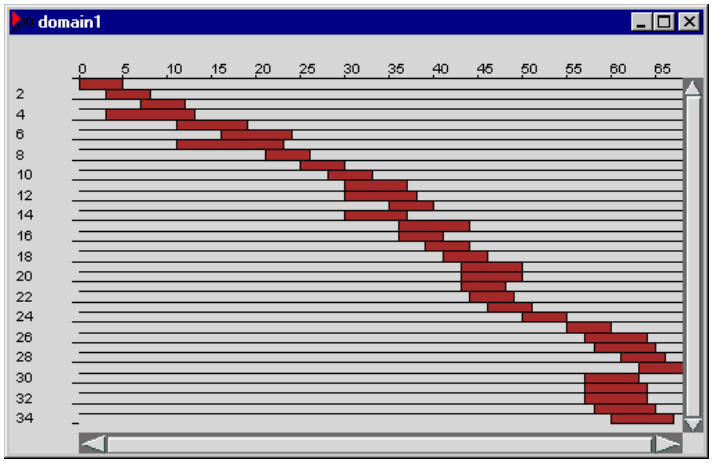

Fig. 12. Initial domains with redundant constraint

constraints. We can make a more detailed analysis looking at the *propagation events* (see chapter **??**) for each constraint, seeing how often a constraint is woken, how often it detects failure or domain reductions. This also applies to global constraints, which always use a combination of different methods to express the declarative constraint. With the propagation events, we can see which methods are used for which type of problem, and which methods are just overhead for some problem instance.

3.4 Discovering structure in the problem

It is often useful to search for some hidden structure in a problem, which we can exploit in a search method. One useful tool for this purpose is the constraint *incidence matrix* in the search-tree tool, which shows all variables on the x-axis against all constraints on the y-axis. Figures 13 and 14 are the incidence matrices of the same problem, but with different initial variable orderings.

The problem we study here is the map coloring problem for the 110 country map from M. Gardner, another classical CHIP example program (15)(11). The figures show that the variable order given by M. Gardner (figure 13) contains a lot of structure, which makes it a good candidate for a static variable selection order. In the randomized order (figure 14), all appearance of order is gone, we would have to discover this order again with our search routine to obtain good results.

3.5 Identifying weak propagation

The last example for the use of the search-tree visualization is the discovery of weak propagation in a program. We had already mentioned in the first example of N-queens that we can find the cause of the failure in the initial search-tree by

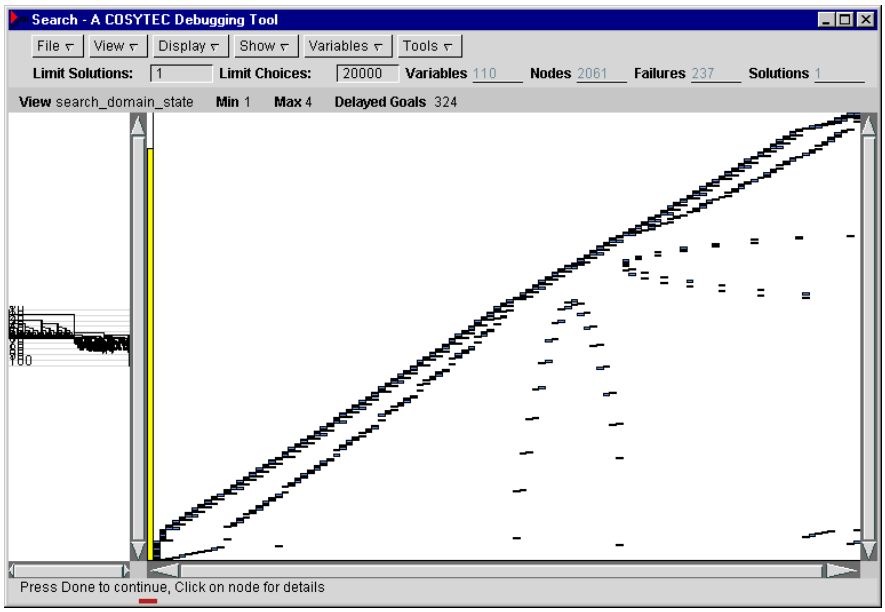

Fig. 13. Incidence matrix with specific variable ordering

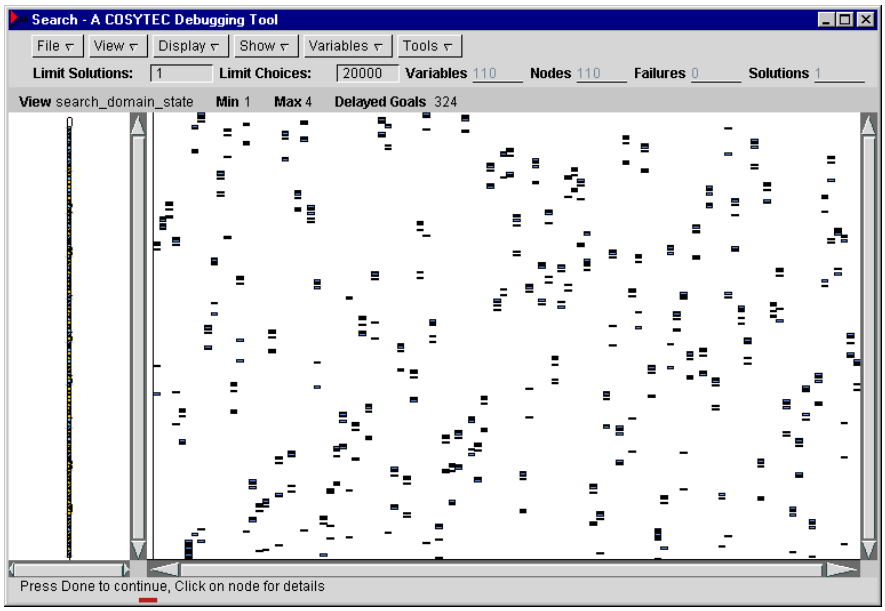

Fig. 14. Incidence matrix with random ordering

counting the variables and possible values remaining. This is a typical situation where we analyze a state of the computation and check manually if there is any missing propagation. Obviously, this requires a good understanding of the problem and of possible propagation methods, even those that are not currently used in the constraint engine. We can try to automate this reasoning by applying a general propagation mechanism to detect if some values can be removed in the domains. *Shaving* (7) is such a general method. After each propagation step, we test each value in the domain of each variable to check if it can still be assigned. If the test assignment fails, we can remove the value from the domain before continuing the search. It is interesting to see if we can come up with a more specific deduction rule that would allow us to remove the shaved values without the expensive testing at each step. Below we show two shaving examples from the square placement problem (1)(4). It overlays the domain display with and without shaving at two steps of the search. The dark colored areas are values which stay in the domain, the lighter colored values are removed by shaving. In the first example (figure 15), the domain reduction is quite insignificant, but in the second step (figure 16) a bit deeper in the search, many values are removed and some additional assignments are detected.

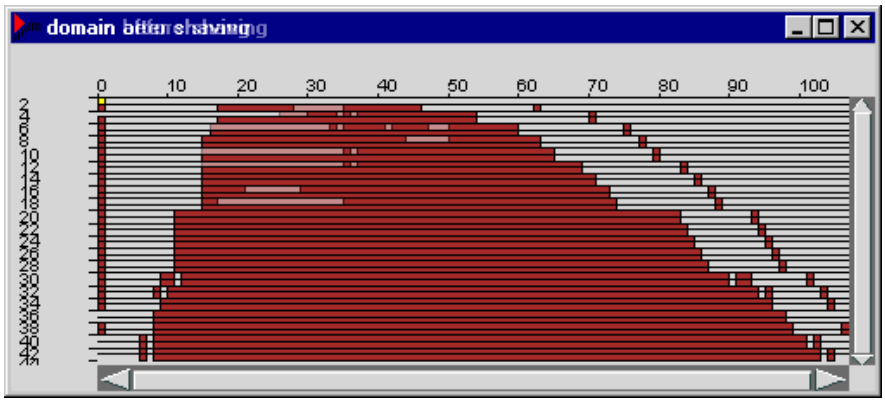

Fig. 15. Shaving at the first step

We may not be able to deduce any new deduction rules, or we may not want to spend the time doing this for some particular problem. In that case we can still apply the shaving directly, if the additional domain reductions balance the cost of the testing operations.

4 Case Studies from Industry

This section describes a number of industrial applications that were checked with the visualization tools for CHIP. The applications had been developed and

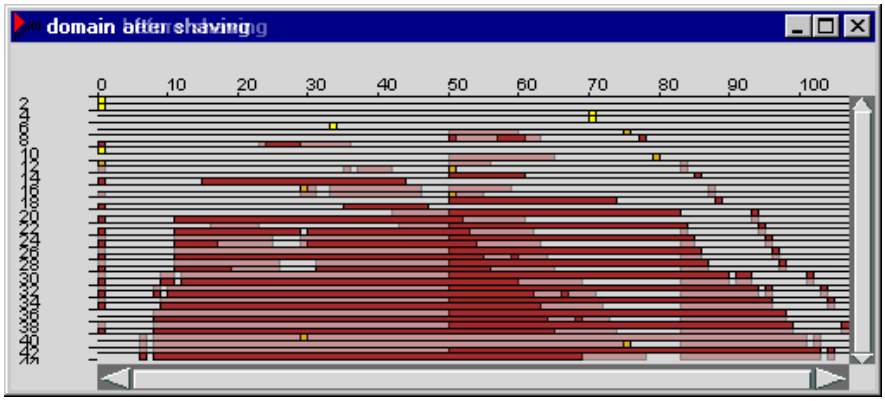

Fig. 16. Shaving operation removing many values

sometimes delivered to the customers before the search-tree tools had been available. As part of the DiSCiPl assessment phase, the applications were analyzed with the tools to discover the usability of the tools, but also to see if improvements to the original programs could be made. The first two applications have been developed by OM Partners, the last two by ICON. More details about the applications and the assessment results can be found in (8).

4.1 Scheduling application involving setup cost

Problem description The problem consists of scheduling a number of tasks (about 100 tasks) on 8 machines. Machines are divided in two classes. Each task has to be scheduled on a machine in a given class, depending on the task type. Moreover, the machines allowed for a task are ordered by preference. Some tasks are already fixed on a particular machine. Most of the tasks additionally use a common resource (laboratory personnel). The number of instances for this resource is limited. Further, each task has a different *bill of material* and the availability of raw material is limited. The schedule also takes into account the unavailability of machines and resources. The end products produced by the tasks belong to different product families. Transitions between different product families in the schedule involve a setup cost. For each task, an earliest and latest production time is specified. The earliest time is a hard constraint, the latest time can be relaxed. The scheduling horizon is 0..50000 minutes (about 34 days). The goal is to find a schedule that minimizes the overall setup cost while at the same time keeping the lateness of tasks under control, also taking into account the preference of machines for each task. A parameter allows weighting the importance of setup cost against the lateness of tasks. As the search space is quite large, the problem is relaxed to finding a good quality solution (sub-optimal) reasonably fast (i.e. within some seconds).
The program already existed before the start of the DiSCiPl project. It includes several `diffn` and `cumulative` constraints (besides equations, inequalities and

`element` constraints). A sophisticated labeling strategy is used to find a reasonably good solution. At the time the program was written, the `cycle` constraint was not yet available. Therefore, reducing setup costs was integrated in the labeling strategy. In the context of DiSCiPl, the program was rewritten using the `cycle` constraint. This constraint allows modeling the sequence of tasks on each machine as a cycle, in which the setup costs act as weights.

Evaluation We use the visualizers `visualize_cumulative_resource` and `visualize_assignment`, combined with the search-tree tool showing the end time and machine variable for each task. In the new version of the program, `visualize_graph_lines` was added to visualize the `cycle` constraint. Adding the necessary annotations for the search-tree tool and visualizers required only minor modifications to the program. In order to use the visualizers, `visualize/4` wrappers are simply put around the `cumulative`, `diffn` and `cycle` constraints. However, some care is needed when the same routine setting up a constraint is called several times (e.g. the same routine may be used to set up a `cumulative` constraint for each resource). In that case, a unique identifier has to be created for each visualizer (based on the information passed to the routine). Using the search-tree tool requires to create the list of search variables and to store some bookkeeping information (in order to obtain the number of the search variable during the labeling phase). Adding the latter information is very easy if the program uses CHIP++ objects. Each task is modeled as an object instance, containing all information about this task. The number of each task for use in the search-tree tool can just be added as an extra data field in the object instance.
Before labeling is started (i.e. just after setup of the constraints), the visualizers already show the position and resource use of fixed tasks (tasks that cannot be moved and dummy tasks used to model unavailability of machines/resources). This allows verifying the correct modeling of fixed parts. Afterwards, one can check how labeling proceeds. For example, the debugging tools allow checking if tasks are considered in the correct order, e.g. if ordering is based on minimal end time. However, when ordering is based on more complex criteria (e.g. setup costs between task types), the tools do not provide much help. One still has to resort to more complex (external) visualization tools (e.g. a Gantt representation where tasks can be colored according to task type). By default, the displayed area in the visualizers is limited by the upper domain limits of the associated variables. This yields a general overview of the problem. But, it is also possible to zoom in on a specific area in order to obtain more details.

The search-tree tool confirms that, in the original program version (search-tree shown in figure 17), the implemented labeling strategy yields a reasonably good solution without backtracking (setup cost: 44, cost of tasks being too late: 3216 $\Rightarrow$ overall cost of 3656). However, the search space is too large to get to the optimal solution. Use of `min_max` with the time-out parameter to limit the search resulted in a (sub-optimal) solution with cost 3390 (setup cost: 26, cost of tasks being too late: 3130), for the given data set. As mentioned above, an alternative

program was developed that uses a `cycle` constraint. The `cycle` visualizer helped in setting up this new model (e.g. helped in identifying the missing propagation due to errors in the modeling and showed the need to use the `origin` parameter of the constraint). The labeling part could be simplified by exploiting the `cycle`: direct labeling of the weights (representing setup costs between tasks) could be used in order to limit setup costs. The remaining labeling code focuses on restricting lateness costs. For the example datasheet, the new version of the program (search-tree shown in figure 18) results in a first solution with cost 3494 (setup cost: 18, cost of tasks being too late: 3314). This is significantly better than the one obtained with the original program as far as setup costs are concerned (the `cycle` leads to a better restriction of the search space).

The original program contained an error in the construction of fixed (dummy) tasks to model the unavailable periods of resources, causing no such tasks to be created. The visualizers immediately showed the absence of these tasks.

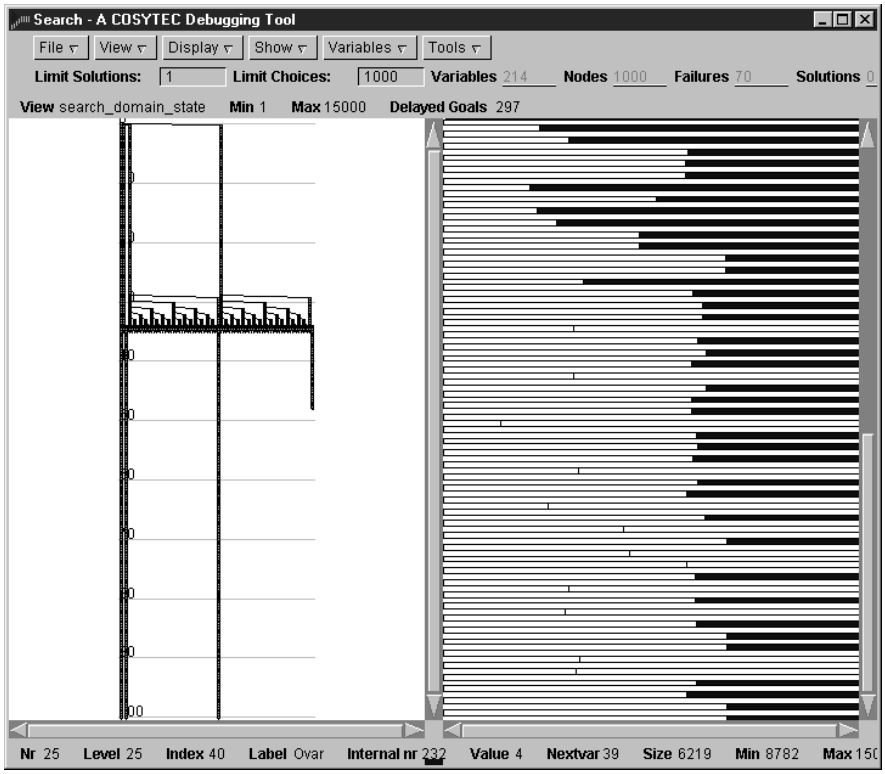

Fig. 17. Original program with min_max

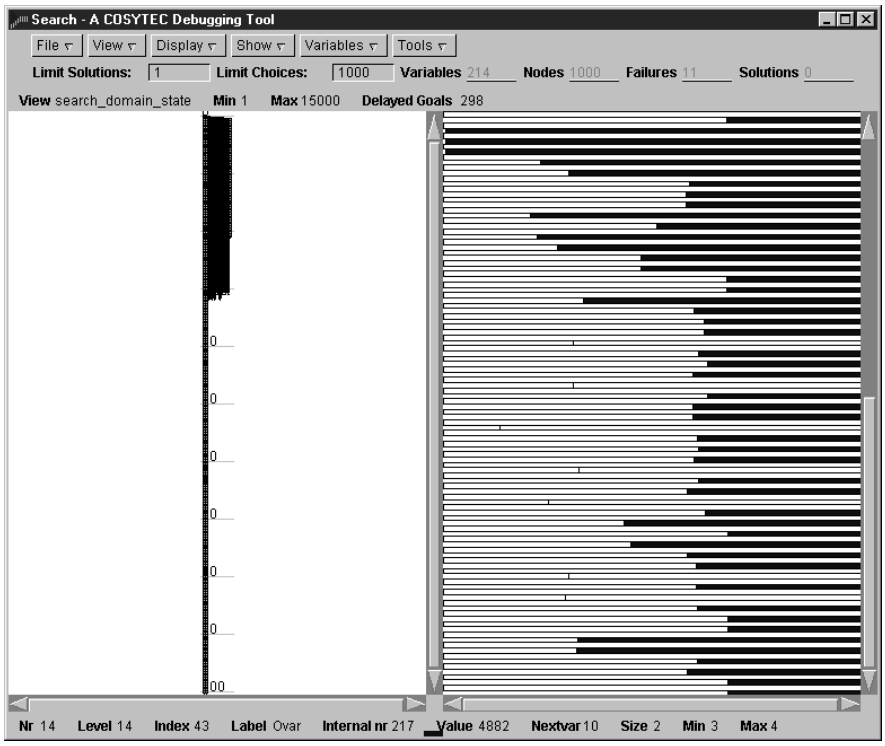

Fig. 18. New program with cycle and min_max

4.2 Tank scheduling application

Problem description The problem described in this section is a tank planning + shipment problem. A task consists of the following steps: creation of a certain product quantity in a reactor, storage of the product within a tank, and loading (shipment) of the product out of the tank using some loading machine. As soon as the reactor step is finished, the product has to be put into some tank (fill action). This tank is chosen among a pool of available tanks. Of course, tanks have a limited size and a tank should not contain different products at the same time. At some point, (a part of) the tank content has to be loaded (empty action). The choice of loading machine depends on the tank. Loading is only possible during working hours. The problem has been solved in different phases: The first goal was to schedule the fill, empty and loading actions, keeping the reactor steps fixed and taking into account all tank and lateness constraints. The program involves several `cumulative` and `diffn` constraints (on tanks and loaders). Alternative formulations are possible to take into account the tank capacity:

- a `diffn` in 3 dimensions (each item in the `diffn` models the storage of a product in a tank, lasting from fill to empty, after the product has come out of a reactor step), or
- a `cycle` using loading and unloading operations (argument 12 of the `cycle` constraint (6); this `cycle` models the sequencing of fill and empty actions on tanks).

The second goal was to also take the reactor planning into account. On the reactor there are major setup times between different product families. As usual, it is difficult to determine the relative importance of setup time versus lateness.

Evaluation (Part I) In this part a solver has been made for the tank assignment. The reactor steps are fixed.
We use the `visualize_cumulative_resource`, `visualize_assignment` and `visualize_graph_lines` tools together with the search-tree tool showing the start time, end time and machine variable for each task.

The `cycle` visualizer shows how fill and empty steps on the tanks are scheduled. E.g. in figure 19 below (note: a fill and its corresponding empty step have subsequent numbers), the sequence on tank 8 (first line in display) is fill1(22)-empty1(21)-fill2(28)-fill3(32)-empty3(31)-empty2(27); the tank is empty after step empty1 (21) and after empty2(27). Unfortunately, the step numbers shown in the `cycle` do not reveal which product is involved. So, it is difficult to check whether a tank never contains a product mix. In fact, the program initially contained a bug with respect to this constraint. This bug was only detected through an external visualization tool (part of our own application) where coloring of tank steps on product type is possible.

The `cycle` visualizer only provides information about the relative sequence of fill/empty steps. It does not indicate their absolute position in time. The latter can be derived from the `diffn` visualizer on tanks. However, a problem is

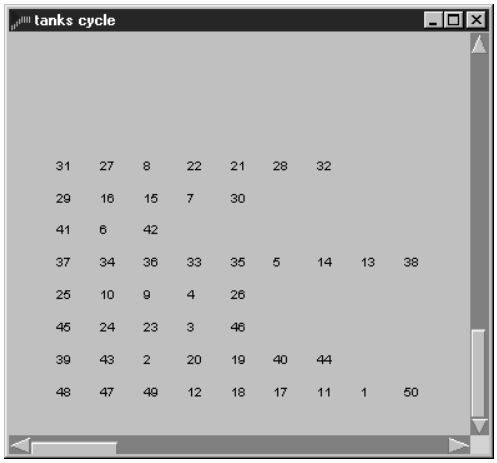

Fig. 19. Cycle visualizer on tanks

that step numbers do not correspond in the different visualizers (see below). A `cumulative` visualizer shows the workload of the loaders. We can see at what times the maximum capacity is reached. The `diffn` visualizer gives more information about which loaders are used at what times.

Evaluation (Part II) The reactor steps have to be scheduled in such a way that the setup time + lateness cost is minimized and that the tank planning (see Part I) is still feasible.

We used additional `visualize_assignment` and `visualize_graph_lines` tools in combination with the search-tree tool.

In this case the `cycle` visualizer and the `diffn` visualizer both show the sequence of the production steps on the reactor. The `diffn` visualizer better reflects the time aspect of the problem, the `cycle` visualizer better reflects the number of tasks already placed. Unfortunately, the steps shown in the visualizers do not reveal which product is involved nor how high the associated setup and lateness cost is. The search-tree view, however, allowed to detect that the node numbers in the `cycle` constraint were not defined correctly. Once the constraints on the reactor scheduling were modeled correctly and the first feasible solution had been found, the search-tree view also was a real help during the optimization of the solution. The format of the search-tree helped the user to detect trashing and showed at which point the program performed (useless) backtracking (see figure 20).

The trashing could be avoided by changing the labeling routine. The users realized that it is not required to try every successor of a node. It was sufficient to try, within the same product family, only the step (i.e. successor) with the earliest due date. The search-tree obtained in the final version of the program is

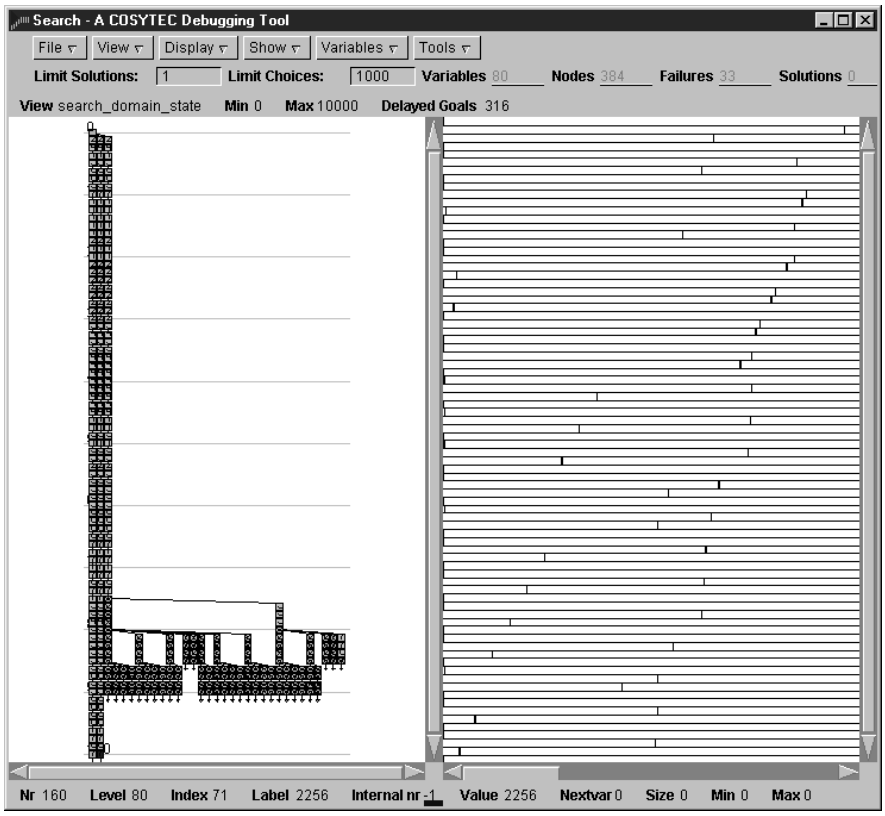

Fig. 20. The (incomplete) search-tree view reflecting useless backtracking (trashing)

shown in figure 21. The ameliorated labeling routine allowed the user to obtain an optimal solution in a reasonable time.

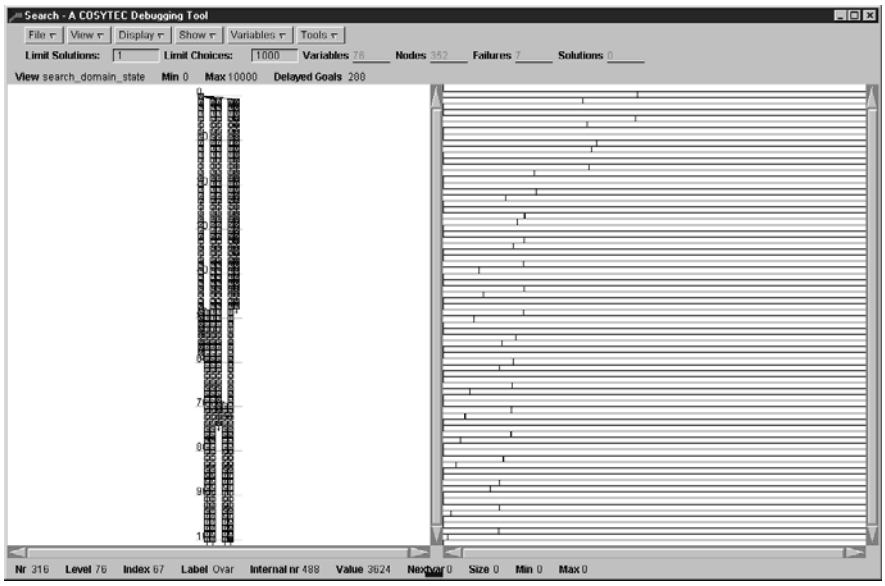

Fig. 21. The complete search-tree obtained for the final version of the program

4.3 Layout of mechanical objects: graph coloring

Problem description The problem consists in the determination of the configuration of mechanical pieces, which must respect several mutual relations. An instance of the problem is given by:

- a set of objects to configure;
- a set of relations to be satisfied.

Configuring an object means setting up its elements. These elements can be disposed with a set of physical assembly constraints. The relation that must hold between the objects (different for each problem instance) is satisfied by the identification of the elements to be included in any object. The problem is divided in two main parts. This section considers the first one, which consists in the identification of the mechanical elements to be inserted in each object, in order to satisfy the set of relations between them. This problem is basically a *graph coloring* problem with a complex specification, where the graph is defined by the set of relations between the objects. The goal is to find a coloring of the graph that minimizes the number of colors used, which is given by a `min_max`

minimization. The domain variable selection strategy is *most constrained*, and the value generation starts from the domain minimum value. The model only uses disequality (`#\=`) constraints. The size of problem instances varies from one to several hundreds of objects and relations. Just as an example of the complexity of the constraint problem, on a small instance with 10 objects and 12 relations, 926 disequality constraints are posted on 65 variables to define the corresponding graph.

Evaluation For this example, we only used the search-tree tool. There were no major problems annotating the program for the search-tree tool. The `min_max` timeout parameter was included in our production application because an optimal solution can not always be found for each instance of the problem. We saw that with the search-tree tool this parameter is useless, and that it can be replaced by the GUI field `Limit Choices`. Attempting to include in the tree the cost variable (which is not assigned in the labeling) required some extra work on the `search_start` call.

With the tool we found a first benefit without any particular intervention on the code, as two facts were immediately visible: the number of generated constraints and the number of variables. These two values vary a lot depending on the problem instance, and characterize well how hard the problem is. The variables/constraints incidence matrix was very useful in finding disequalities (`#\=`) on variable pairs that were duplicated during the constraint generation phase.

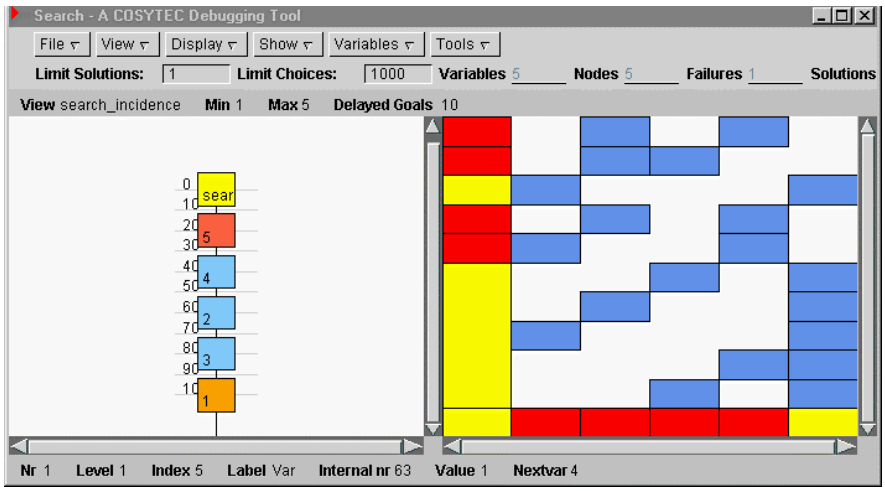

Fig. 22. Layout first phase, incidence matrix

This problem was already recognized during the original code development (of course without the search-tree tool), and is due to the complexity of the gen-

eration of constraints for an instance of the problem. The number of duplicated constraints did not seem big enough to significantly influence the computation; thus this problem was ignored during the development. But this possible optimization of the program was easily discovered with the search-tree tool (at least for small instances) by checking the incidence matrix (figure 22). With the tool we verified that the implemented search strategy is useful to quickly find good solutions. However, even for medium sized problems the optimal solution requires the exploration of wide trees. As an example, the next picture (figure 23) shows a tree exceeding over 5000 nodes (restricted with the tool parameter `Limit Choices`).

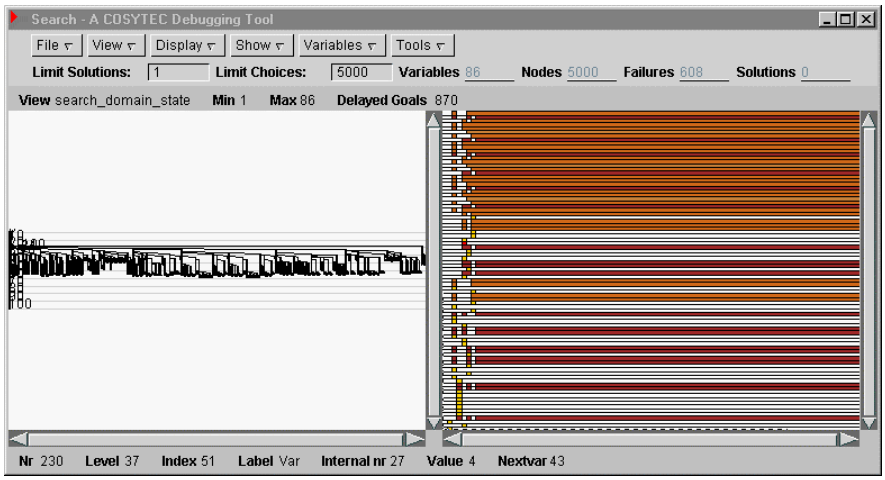

Fig. 23. Layout first phase, domain state, limited search

During the analysis of small problems with the search-tree, we discovered that the labeling does not avoid the generation of solutions that are permutations of each other. In order to recognize this behavior, we manipulated the tree view by collapsing some branches. These branches are equivalent, as after the first domain variable assignment they lead to the same propagation, differing only for a permutation of the assigned values. In figure 24, the two solutions found are shown, each with the same value for the first variable. In addition, the first failure branch with the same first variable (index 39, value 1) is expanded. The search for a solution could stop even after the first failure branch exploration, as it represents the optimality proof of the previous solution: the following sub trees are equal to the expanded one, except for the assignment order; for instance, the next branch starts with the variable 39, value 2.

This problem was considered even in the development phase, but the need for a quick near-optimal solution led to a timeout-limited search without making the generation function more complex. The actual generation is based on the

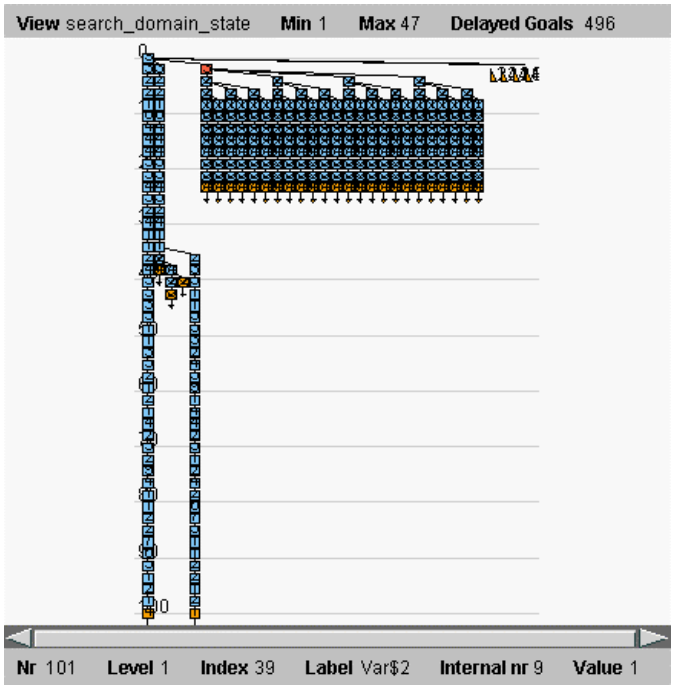

Fig. 24. Layout first phase, first solutions found

selection of most constrained variable, which is given the minimum possible value with `indomain/1`.

```
...
search_start(Ls,min_max(outlab_stv(Ls,Cost),
                        Ls, 0, Num, 0)),
...

outlab_stv(Ls,Cost):-
        maximum(Cost, Ls),
        search_number(Ls,Merge),
        labeling(Merge, 1, most_constrained, assign_stv).

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

On average this strategy seems to lead quickly to a solution that uses few colors (therefore frequently optimal, even if the proof requires a deeper exploration of the search-tree). We modified the generation function to avoid the permutation effect described above. The labeling function then became:

```
...
search_start(Ls,min_max(delete_new(Ls,Cost,[0]),
                        Ls, 0, Num,0))
...

delete_new([],_,_).
delete_new([H|T],Cost,CostP):-
        maximum(Cost,[H|T]),
        delete(t(X,N),[H|T],R,1,most_constrained),
        search_node(X,N,assign1(X,CostP,CostPNew)),
        delete_new(R,Cost,CostPNew).

assign1(X,C,C):-
        C\=[0],
        member(X,C).
assign1(X,C,[H1|C]):-
        C=[H|_],
        H1 is H +1,
        X=H1.
```

With the search-tree tool it was then possible to see a reduction in the tree. Unfortunately, we noted that the instances of the problem vary greatly. Consequently, the same generation function does not show a uniform behavior; i.e. it is not always better than the others, at least when measuring execution time.

4.4 Layout of mechanical objects: physical layout

Problem description This section considers the analysis of the second phase of the mechanical object layout application described previously. Given the set of objects to configure, the second part of the problem is aimed at the physical layout of the elements in each object, meeting a fixed set of physical constraints (which are invariant for any instance of the problem). In this phase there is no need of an optimal solution, we are satisfied with the first solution which properly places the elements identified by the first phase. This stage involves the use of a `cumulative` constraint to define the physical constraints that mechanical elements must respect. Other used constraints are `among`, `element` and `alldifferent`.

Evaluation We used the `visualize_cumulative_resource` tool together with the search-tree tool.

The goal of the physical layout phase is to find a placement for the mechanical elements. For any object there are two `cumulative` constraints that express the physical assembly limitations. Each object element is represented with a task that uses one resource. Its "duration" is the space that could cause a placement conflict if owned by another element. The `among` constraint captures the optional distribution of the elements in particular areas of every object. The first analysis was run on a small example consisting of 7 objects and 7 relations. This way all the visualizer windows could be seen together with the search-tree.

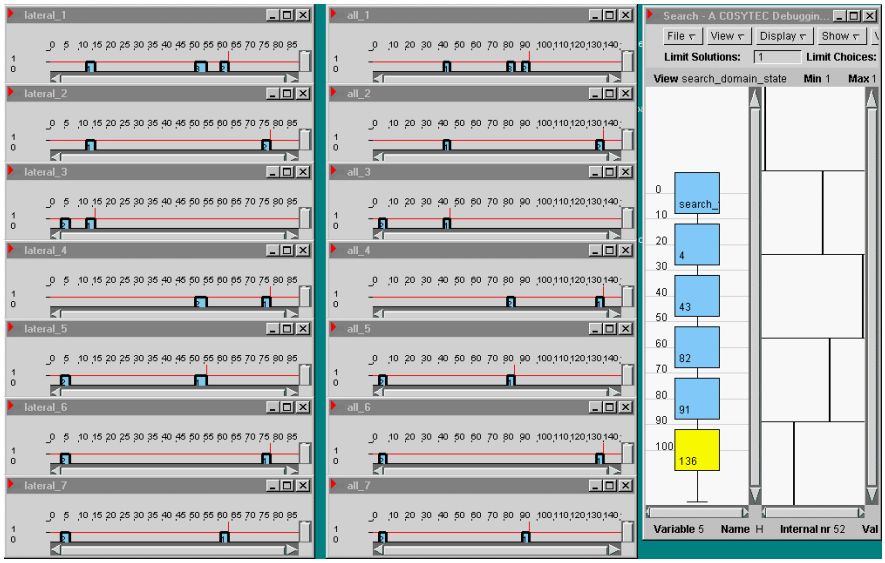

Fig. 25. Layout second phase: lateral_x, all_x and search-tree windows

For any object there were two paired windows (named lateral_x and all_x), representing the two physical constraints. As we can see in figure 25, there were five elements to place (corresponding to the nodes in the tree) that could be joined differently within the objects. For instance, in the all_1 window there were three elements, whereas in the all_3 window there were two. A graphic representation of the element positions is interesting, even if the `cumulative_resource` representation does not completely match our application domain. We can verify on the search-tree that for small problems there is no backtracking at all when the variables are assigned values that are compatible with the physical constraints and there are no additional `among` constraints. If an optional distribution is added, the original labeling initially leads to assignments that do not satisfy to the added `among` constraint (figure 26).

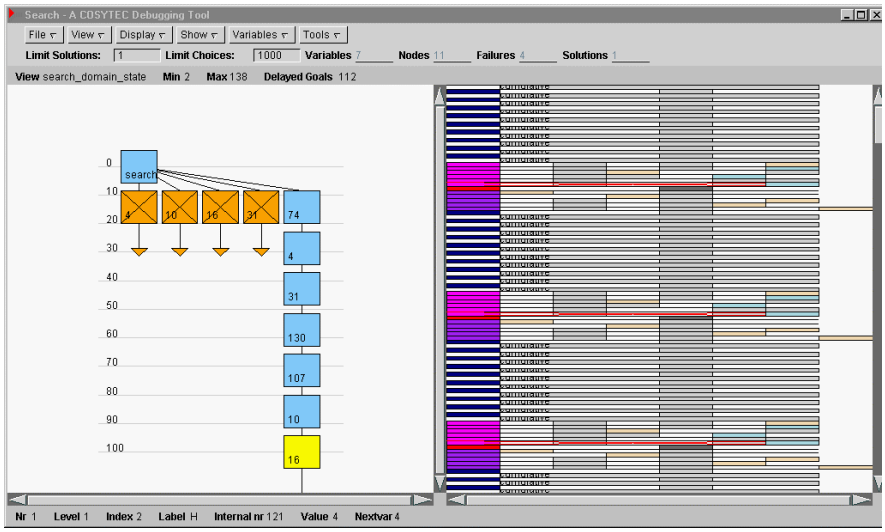

Fig. 26. Layout second phase: failures due to among

In order to detect the reason for these failures, the propagation view is quite helpful. For many problem instances we can see that there is no solution at all, due to the extra constraint `among`. For a larger data-set, we could not identify the reasons for failure, mainly due to the large number of constraints to visualize and the difficulty of selecting a good subset to analyze. On smaller problem instances, a detailed analysis of the propagation view proved to be useful, since it revealed a large number of inappropriate constraint calls and useless propagation (see figure 27):

With the incidence matrix view (figure 28), we could find the excessive number of `element` constraints. It is worth noting that the piece of code responsible for this behavior was developed a long time ago by other programmers, who do

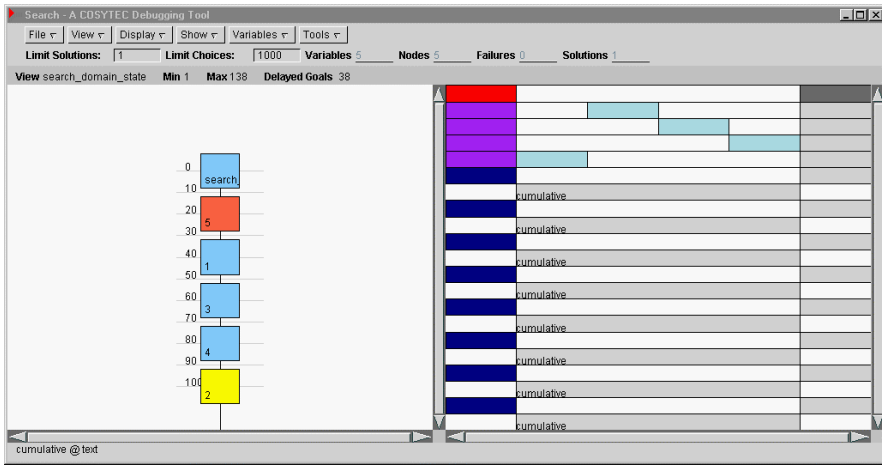

Fig. 27. Layout second phase: Propagation view

no longer maintain this application. Identifying this anomaly would have been difficult with standard debugging techniques.

Hence, with the search-tree tool analysis the constraint generation anomalies were fixed. Specifically, even if there were no modeling errors, we modified the generation of the element constraints to speed up their definition. The propagation and propagation events analysis was quite complex, but it allowed us to understand the actual CHIP behavior when waking the constraints at a detailed level.

5 Analysis and Possible Improvements

All programmers started using the debugging tools on academic examples and course exercises before experimenting with existing and new industrial applications. The different users focused on different parts of the debugging tools: ICON concentrated more on the CHIP graphical propagation views, OM Partners concentrated more on the CHIP global constraint visualizers.

5.1 Need for a debugging methodology

For the tool developers the results obtained with industrial applications are certainly most interesting. However, a lot of time had to be spent learning how to interpret the information shown by the visualization tools. Although it is rather easy to get the graphical debugging tools working, the users strongly recommend the development of a *graphical debugging methodology*. This debugging methodology could help the end user to exploit the visual information in the best way: which conclusions can be deduced immediately from the search-tree geometry and/or domain behavior? What views are most interesting when going into more detail? Which functionality still has to be discovered?

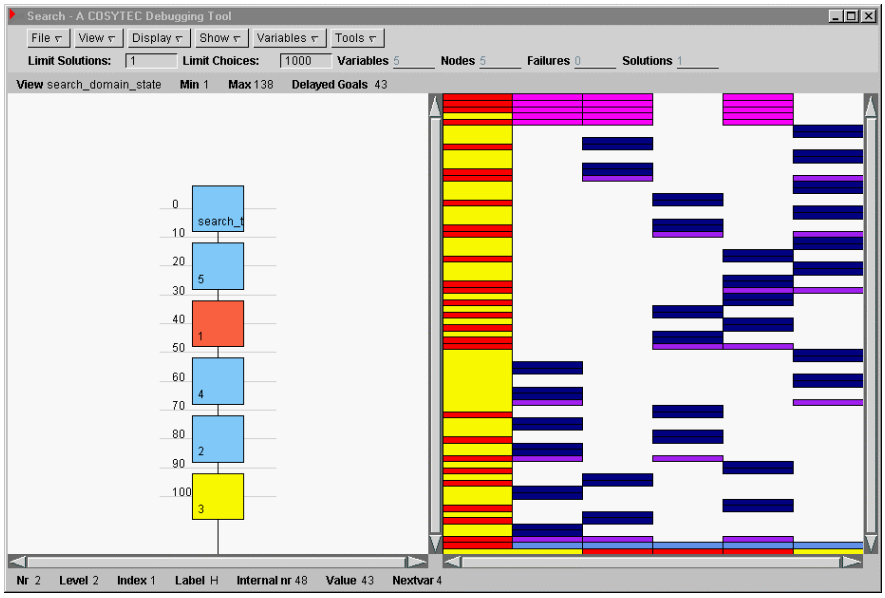

Fig. 28. Layout second phase: Incidence Matrix view

5.2 Program improvement due to the use of debugging tools

Before starting the assessment phase, it had been planned to evaluate the debugging tools by using statistics on the number of errors, the developing time, etc. when programming with/without debugging tools. For several reasons this was not feasible: First, given the time and resource limits of the project, industrial partners could not afford to train people or to develop new applications without using the best and most advanced tools. Next, since the graphical debugging tools only can be used when a program already runs, the debugging support for a large part of the program development has not changed. We still have:

- The typical syntactic and semantic errors. In general, they can be found using classical debugging techniques like text tracing. Here, the assertion tools could give assistance (see chapter **??**).
- The errors due to data inconsistency. These inconsistencies often cause the violation of constraints and the immediate failure of the CP program (the famous "no" answer). The detection of these data inconsistencies is very time consuming and can in general not be done by simple text tracing.

Consequently, objective before/after comparisons were only possible when using the debugging tools in existing applications and looking to areas where the graphical debugging tools provide assistance, for example:

- the behavior and efficiency of the labeling/optimization routine,
- the efficiency of the constraint propagation,

– the effect of adding redundant constraints to the model.

All users have found that *all existing industrial applications for which the debugging tools have been used have been improved* (see detailed application descriptions of OM Partners, ICON and PrologIA in (8)). In most cases the labeling routine and/or propagation have been improved and result in a performance amelioration and/or better optimization. However, the number of treated applications is not sufficient yet for detailed statistics.

5.3 General conclusions on the CHIP graphical debugging tools

The graphical representation of CLP problems through the search-tree tool and global visualizers clearly is what a user needs to understand and explore different search strategies. They encourage the user to optimize the search strategy and to go farther than the first feasible solution. Although most real-life problems still have to be scaled when using the debugging tools, the number of variables and constraints that can be used is big enough to test valuable industrial cases. As a result, new applications are written in such a way that the program can be started with/without debugging tools. The expert users are convinced that using the debugging tools will make the maintenance of industrial applications easier and less time consuming.

search-tree tool The overall view (shape of the search-tree) in the search-tree tool immediately gives an intuitive picture of how well the search strategy works (what is the degree of propagation). Afterwards, the user can inspect the execution in more detail, by clicking on the nodes of interest (comparing different nodes) and investigating the different views. The search-tree tool allows complete navigation through the tree, not just between adjacent nodes. Novice users reported that the constraint paradigm became immediately clear, expert users realized that a lot of difficult text tracing work, trying to understand what was happening, will be avoided in the future. For all users, the domain state and update views in the search-tree tool are quite easy to understand but working out the propagation view requires some more effort. The propagation event view is even less accessible for an end-user, but may no doubt provide essential information to the implementers of the system. Some suggestions for an easier and better search node annotation are given below. Keeping track of the search node number requires minimal effort when using CHIP objects (in that case the number is just an extra field in the object structure); it is not necessary to extend or build data structures to link the numbers with the variables. However, especially when dealing with large problems, the user should have the possibility to define an application specific node name.

Global constraint visualizers The global constraint visualizers offer graphical representations that are tailored to the specific problem at hand (to a specific use

of the constraint). The fact that they can be used in combination with the search-tree tool results in a powerful tool set. Adapting the program to use the global visualizers was straightforward, as it only involves adding a simple wrapper around the global constraints. Combining the global constraint visualizers with the search-tree tool has the advantage that the visualizers are automatically updated when navigating through the search-tree. However, it also requires that all variables determining items (e.g. rectangles) in the visualizers are included as search-tree nodes. These extra search nodes complicate the views in the search tool, mixing up the relevant information with irrelevant details. Standalone use of the visualizers may provide more information than in combination with the search tree tool, e.g. if the predicate `min_max` is involved. The reason is that the search-tree tool currently does not remember constraints introduced during the search. The graphical user interfaces (GUI) of the global visualizers often come very close to dedicated application GUI. Although it is not the intention to compete with specialized GUI, it should be possible to extract more user defined information from the global visualizers. Suggestions for improvements can be found below.

5.4 Suggestions for extensions to the CHIP debugging tools

User definable annotation In the current tools, the annotation of the search nodes and items in the visualizers is predefined. Especially when dealing with large problems, this information is quite cryptic for the user. It is difficult to relate the annotation with the problem concepts (e.g. tasks to be scheduled). Moreover, there is no uniform notation (numbering) of variables/objects over the different visualizers and search-tree views. Hence, clicking on some item (rectangle) in a particular visualizer highlights items with corresponding numbers in other visualizers, but these do not necessarily correspond to the same task. A possible extension is to allow user-defined annotation. In this way, the user can specify information that is relevant for the specific problem at hand:

- In the search-tree tool, the `search_node` predicate could be extended such that the user can specify the node label, e.g.

  ```
  search_node(X, N, 'task3-product1', indomain(X))
  ```

  An extra option "Show User Label" in the search tool could then annotate the search node with this label.
- The visualize wrappers could also be extended to allow user-defined annotation of the items in the visualizers. E.g. an extra argument could be the list of labels corresponding to the list of rectangles in the `diffn`, or the list of variables (starts, duration, ends,..) in the `cumulative`.

A further extension would be user-defined coloring of search nodes or visualizer items based on some characteristic of the underlying problem concept (e.g. based on task type). Based on these annotations the user could identify the search-tree nodes that are interesting according to a certain criterion, and consequently decide which part of the tree to observe.

Tree information

- It would be interesting if the user could easily identify on the tree some areas sharing common characteristics, for instance big domain reductions, isomorphic sub trees, unaccounted problem symmetries, etc. The availability of statistics could be quite helpful to let the user compare the quality of different strategies.
- Another possible improvement is to highlight/color the nodes that (do not) satisfy specific conditions. For instance, during performance debugging, the user should be able to quickly identify `min_max` failure nodes (i.e., solutions worse than a previous one) in order to refine the heuristic. With the present tool, the user must select any failure node and find out if the cause of failure is interesting.
- No information is provided about *shallow backtracking*, i.e. backtracking within a search-tree node. Such information is also relevant when comparing different labeling strategies and their degree of propagation. A proposal is to display the number of backtracks within each search-tree node.
- Constraints introduced during the search are currently not remembered by the search-tree tool (e.g. the extra constraint introduced by `min_max` after a first solution has been found). This results in a wrong image in the domain state view (the domain may actually be smaller than shown). The correct domain can only be seen in the indication of the domains in the search nodes.

View manipulation utilities The user would be helped during both correctness and performance debugging if the tool gave some assistance in the search-tree exploration. Some possible improvements could be the following ones:

- Collapsing the nodes that represent variables becoming ground because of propagation. Actually, a typical event is the reduction of variable domain to a single value, due only to constraint propagation following some assignment. This fact is translated on the search-tree into node chains where there is no additional domain reduction. The graphical representation of this property would avoid that the user analyses these nodes, which do not carry any extra information.
- Defining variable order in search-tree views. It would be helpful to let the user change the order of variables in views, in order to have a better visual representation for certain problems without tweaking with the `search_number` adornment. More generally, it would be advisable to simplify the search-tree tool wrapper, in order to avoid the reordering of traced variables inside the code (see above). Another solution could be to allow the user to dynamically change the visualization and ordering of the elements inside the tool with mouse actions, for instance by means of drag-and-drop of rows and columns, or with pop-up menus.

New views

– User definable representations (based on existing view dimensions) The 2D representation of domains is suitable. As a possible extension, we suggest a more general framework where the user defines his/her own representations. The idea is that if the tool provides some "dimensions" (e.g., list of the variables, list of the domains, constraints, propagation, etc.), the user could select as horizontal or vertical axis the preferred items, possibly selecting subsets or organizing them in parallel layers, in order to represent/visualize also multidimensional problems.
– A utility to compare graphical views pertaining to different nodes. Very often during the analysis it is important to understand the ongoing domain reduction. In the current tools, the user selects in turn two different nodes in order to depict the changes. It would be better to have a "diff" utility that represents graphically the differences in a view between two nodes (e.g., a "Diff Domain State" view).

Extension of the global constraint visualizers

– Standalone visualizers: breakpoint with user interaction. During standalone use of the visualizers, the user should have the possibility to insert a breakpoint upon which some action can be taken. E.g. it would be nice if the user could ask for additional information, such as the domain of a variable, etc. Currently, one has to decide in advance which information has to be displayed. No interaction / extension is possible once program execution has started.
– 3D diffn_visualizer. A visualizer of 3D problems could help in case of bin packing problems.
– Better scrolling and zooming functionality. The current zoom facilities do not change the precision of the scales when zooming to a small part of the original range.

6 Conclusions

In this chapter we have described some practical experience with the use of the CHIP visualization tools. We have presented how they can be used to detect and overcome typical problems of performance debugging. These rather general rules are a first step to a more refined methodology of performance debugging with visualization tools. In a second part, we have presented a number of industrial applications that were checked during the assessment phase of the DiSCiPl project. These applications had been developed before the tools were available, but in each instance some improvement of the program was discovered by using the different tools.
The current state of the visualization tools is just a beginning. In the last section of the chapter, we present some comments on the existing tools and ideas for their improvements. Clearly, the existing tools are already very useful for developing industrial constraint applications, but more work is still required to make

them more accessible for programmers and to allow an easier interpretation of results.

Bibliography

[1] A. Aggoun, N. Beldiceanu. Extending CHIP in Order to Solve Complex Scheduling Problems, Journal of Mathematical and Computer Modeling, Vol. 17, No. 7, pages 57-73,Pergamon Press, 1993.

[2] N. Beldiceanu, E. Bourreau, P. Chan, D. Rivreau. Partial Search Strategy in CHIP, 2nd International Conference on Meta-heuristics, Sophia-Antipolis, France, July 1997.

[3] N. Beldiceanu, E. Bourreau, D. Rivreau, H. Simonis. Solving Resource-Constrained Project Scheduling Problems with CHIP Fifth International Workshop on Project Management and Scheduling, Poznan, Poland, April 1996.

[4] N. Beldiceanu, E. Bourreau H. Simonis. A Note on Perfect Square Placement. COSYTEC Technical Report, January, 1999.

[5] N. Beldiceanu, E. Contejean. Introducing Global Constraints in CHIP, Journal of Mathematical and Computer Modelling, Vol 20, No 12, pp 97-123, 1994.

[6] E. Boureau. Traitement de Contraintes sur les graphes en programmation par contraintes, PhD thesis, L.I.P.N. Universite Paris 13, March 1999.

[7] Y. Caseau, F. Laburthe. Cumulative Scheduling with Task Intervals. Proceedings of the Joint International Conference and Symposium on Logic Programming, M. Maher (Ed), The MIT Press, 1996.

[8] T. Cornelissens. General report on assessment of the tools. DiSCiPl deliverable D.W.WP1.3 M1.3, April 30, 1999.

[9] I.P. Gent and T. Walsh. CSPLIB: A Benchmark Library for Constraints. In J. Jaffar (Ed), Principles and Practice of Constraint Programming CP99, Alexandria, VA, October 1999, pages 480-481.

[10] M. Fabris et al., CP Debugging Needs and Tools, In Proc. Of the 3rd. Intl Workshop on Automated Debugging-AADEBUG97, Pages 103-122, Linkping, Sweden, May 1997.

[11] M. Gardner. Scientific American, April 1975.

[12] ROSEAUX. Exercices et Problèmes Résolus de Recherche Opérationnelle - Tome 3, Paris, 1983, p.279-282.

[13] H. Simonis, A. Aggoun. Search Tree Visualization. COSYTEC Technical Report, DiSiPl deliverable D. WP3.1.M1.1-2, September 1997.

[14] H. Simonis. Visualization in Constraint Logic Programming. Invited Tutorial. PACLP 99, London, UK, April 1999.

[15] P. Van Hentenryck. Constraint Satisfaction in Logic Programming. MIT Press, Boston, Ma, 1989.