Visualization in Constraint Logic Programming

H. Simonis
COSYTEC SA
4, rue Jean Rostand
F-91893 Orsay Cedex
France
Helmut.Simonis@cosyttec.com

Abstract

Visualization tools can be an important help in the design, development and analysis of constraint programs. In this tutorial we present, on several classical examples, the interest and possibilities of visual tools. We introduce different methods of graphically presenting constraints and constraint propagation. This approach is in particular useful for high-level constraint models built from global constraints. Used together with visualizing tools for variables, they can be used to understand the details of constraint propagation methods and the interaction of different constraints. Another type of visual tool can be used to represent and analyze the search tree of a constraint program. We present typical scenarios on how to find and correct problems in the search procedure or the constraint propagation, as well as selecting good meta-heuristics based on the observed form of the search tree.

Table of Contents

1 VISUALIZATION TECHNIQUES 5

1.1 INTRODUCTION 5
1.2 RELATED WORK 5
1.3 PRINCIPLES OF OPERATION 6
1.3.1 Program structure 6
1.3.2 Symptoms 7
1.3.3 Operating mode 7
1.3.4 System Requirements 8
1.4 INTERFACE 8
1.5 VIEWS 10
1.5.1 Types of Views 10
1.5.2 Tree View 10
1.5.2.1 Nodes 11
1.5.2.2 User Interaction 12
1.5.3 Variable Views 12
1.5.3.1 Update View 13
1.5.3.2 State View 14
1.5.4 Constraint Views 14
1.5.4.1 Incidence Matrix 14
1.5.4.2 Update View 15
1.5.4.3 Constraint Count 16
1.5.5 Propagation View 16

2 N-QUEENS 18

2.1 PROBLEM 18
2.2 MODEL 18
2.2.1 Binary Constraints 18
2.2.2 Alldifferent constraints 18
2.2.3 Search Method 19
2.3 PROGRAM 19
2.4 SOLUTION 21
2.5 SEARCHTREE 22
2.6 ANALYSIS 22
2.7 COMPARISON 24
2.8 PHASELINE 25
2.9 OVERLAY 26
2.10 VARIABLE SELECTION 26
2.11 VALUE CHOICE 28
2.12 CREDIT BASED SEARCH 29
2.13 COMBINATION 30
2.14 SUMMARY 30
2.14.1 Simple Problem 30
2.14.2 Small changes have big impact 30
2.14.3 Improvements 31

3 MAP COLORING 32

3.1 PROBLEM 32
3.2 MODEL 32
3.3 PROGRAM 32
3.4 SOLUTION 34
3.5 SEARCHTREE 35
3.6 ANALYSIS 35
3.6.1 Slow to find first solution 35
3.6.2 Value choice too predictable 35
3.7 DOMAIN UPDATE 36
3.8 INCIDENCE MATRIX 37
3.9 SELECTION 39

3.10 SELECTION 40
3.11 VALUE CHOICE 41
3.12 COMBINATION 42
3.13 COMBINATION 43
3.14 REDUNDANT CONSTRAINTS 44
3.15 SUMMARY 46
3.15.1 Simple map coloring problem 46
3.15.2 Small changes have big impact 46
3.15.3 Visualization helps to understand 46

4 SHIP LOADING 48

4.1 PROBLEM 48
4.1.1 The Objective 49
4.2 MODEL 49
4.2.1 The variables 49
4.2.2 The precedence constraint 49
4.2.3 The resource constraint 49
4.2.4 The search method 49
4.3 PROGRAM 49
4.4 SOLUTION 50
4.5 SEARCHTREE 52
4.6 ISOMORPHIC SUBTREES 53
4.7 VALUE CHOICE 54
4.8 OPTIMIZATION 55
4.9 PHASELINE 56
4.10 CUMULATIVE VIEW 57
4.11 REDUNDANT CONSTRAINT 58
4.12 INITIAL DOMAINS 59
4.13 COMBINATION 60
4.14 SUMMARY 61
4.14.1 Small Scheduling Problem 61
4.14.2 Optimization variants 61
4.14.3 Improvements 61

5 CUTTING STOCK 62

5.1 PROBLEM 62
5.2 MODEL 63
5.3 PROGRAM 64
5.4 SOLUTION 65
5.5 SEARCHTREE 66
5.6 ANALYSIS 66
5.6.1 Bad Initial Solution 66
5.6.2 Backtracking before initial solution 67
5.6.3 Deep Backtracking 67
5.6.4 Implication from Cost 67
5.6.5 Rediscovery 67
5.7 OPTIMIZATION 67
5.8 REDUNDANT CONSTRAINTS 68
5.9 ALTERNATIVE 70
5.10 PROGRAM 70
5.11 SYMMETRY REMOVAL 72
5.12 LABELING ON COST 73
5.13 DOMAIN SPLITTING 74
5.14 COMBINATION 75
5.15 SUMMARY 76
5.15.1 Cutting stock problem 76
5.15.2 Improvements 76

6 PRODUCER/CONSUMER 77

6.1 PROBLEM 77

6.1.1 Introduction *77*
6.1.2 Problem Description *77*
6.2 MODEL 79
6.3 PROGRAM 79
6.4 SEARCHTREE 80
6.5 ANALYSIS 80
6.5.1 Search tree *80*
6.5.2 Initial domains *81*
6.5.3 Conclusion *81*
6.6 RESTRICT 81
6.7 RESTRICT ANALYSIS 82
6.8 ALTERNATIVE MODEL (1) 82
6.9 ALTERNATIVE SEARCHTREE 84
6.10 ALTERNATIVE MODEL (2) 84
6.11 PROGRAM (2) 85
6.12 ALTERNATIVE SEARCHTREE (2) 86
6.13 ALTERNATIVE MODEL (3) 86
6.14 PROGRAM (3) 87
6.15 ALTERNATIVE SEARCHTREE (3) 88
6.16 CONSTRAINT DISCOVERY 88
6.17 PROGRAM (4) 89
6.17.1 Alternative searchtree (4) *90*
6.18 SUMMARY 90
6.18.1 Standard model not sufficient *90*
6.18.2 Region model not useful *91*
6.18.3 Placement model ok, but not perfect *91*
6.18.4 No backtracking is not enough *91*
6.18.5 Use of search tree tool *91*
6.18.6 Think small *91*

7 ACKNOWLEDGMENT 91

8 BIBLIOGRAPHY 91

1 Visualization Techniques

1.1 Introduction

In recent years, a significant number of applications have been developed using constraint programming (CP) technology [SIM95] [SIM96] [WAL95] [JM94]. 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 DiSCiPl project [FAB97], 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 [AB93] [BC94] 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 tutorial discusses the search tree visualization tool for CHIP [SIM95A] developed at COSYTEC in the DiSCiPl project in the context of several classical example problems.

The tutorial is structured as follows: In the remainder of chapter 1, we discuss the festures of the search tree tool for CHIP. In the chapters 2-6, we look at classical constraint problems and disucss some performance debugging issues with the help of visualization tools. For each example, we present a simple constraint model and its CHIP program. Analyzing the behaviour of the program, we suggest possible improvements and check for their efficiency. The improvements are well-known techniques in the folklore of constraint programming, the interest here lies in understanding which techniques work for which type of problem and how to predict this behaviour.

1.2 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 [MEI95] 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 [SCH97], 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 at the moment not contain 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 [JON96]. An early paper of Held and Karp [HK71] 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 [BDD97].

1.3 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.

1.3.1 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 [BBC97] 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 constraint part of an application, the most interesting aspect is the debugging of the search process, on which we will concentrate in the current paper. 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.

1.3.2 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.

1.3.3 Operating mode

We rejected the use of an off-line tool for visualization, although these have been quite successful in visualizing logic programming languages [CGH93]. 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

We can thus ensure a result, even if the user defined routine does not terminate. 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 choices in the search are admissible. In the following, we assume a value choice.

1.3.4 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 is useful.
- 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 very useful for statistics and meta-programming.

1.4 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 indicating individually the variables 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 annotation of built-in search procedures using the predicate **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 to provide this functionality.

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 all non-deterministic parts 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 width and depth of the search tree. The predicate **search_node/3** takes three arguments, the variable which is affected, the index of that variable 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 [AB93]. The interface to the search tree tool is shown in bold.

```
?-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)).
```

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 [SIM95]. 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.

1.5 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.

1.5.1 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 individual nodes. This concept is used for advanced features like propagation lifting, described at the end of the paper.

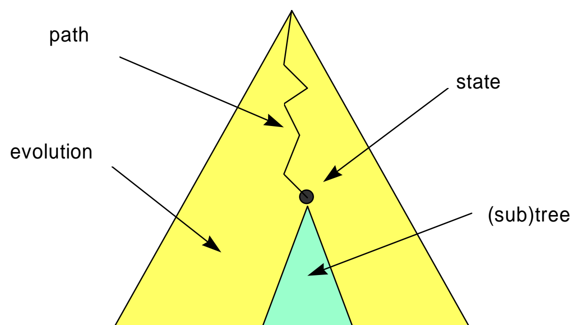


Figure 1: View concepts

- **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.

1.5.2 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.

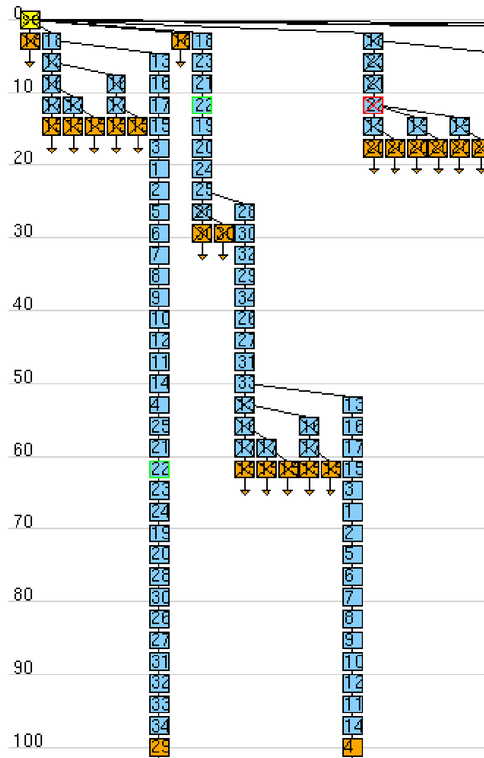


Figure 2: Tree view

1.5.2.1 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 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.

1.5.2.2 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.

1.5.3 Variable Views

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

1.5.3.1 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 4, showing the update view for the first success node of the search tree.

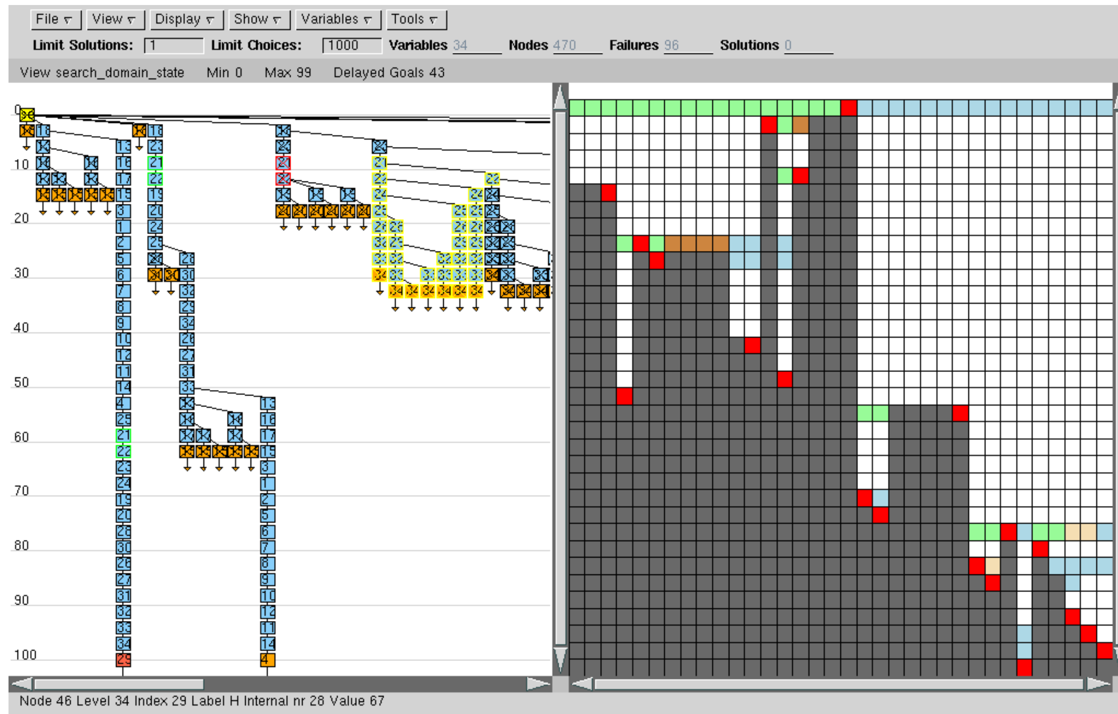


Figure 3: Variable Update View

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

- variable is set in this step by search procedure
- variable is ground
- min and max are updated
- min is updated
- max is updated
- size is changed, i.e. interior value was removed

Selecting a cell in the view will show the level and the selected variable in the information display line. This view gives a good indication 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.

1.5.3.2 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 5 shows the variable state view after the first variable has been assigned.

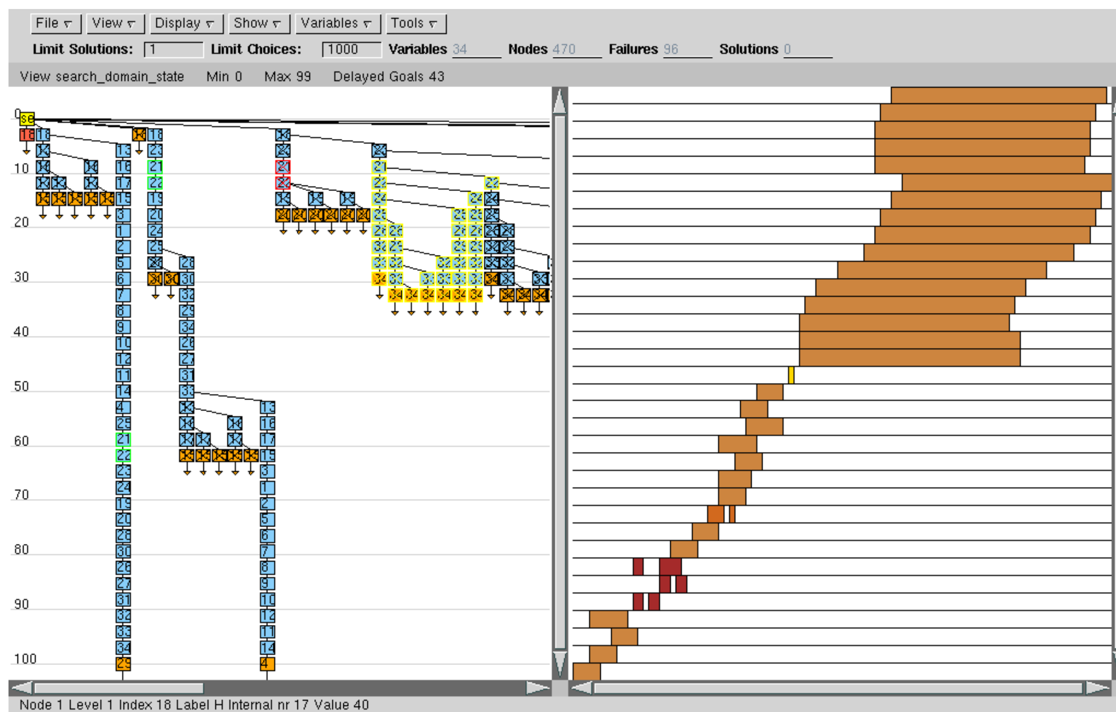


Figure 4: Variable state view

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 use it for example to understand which values are used more often than others.

1.5.4 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. Some of the constraint views can be used for all types of constraints, others are specialized for the global constraints in CHIP. We only discuss the general views in this paper.

1.5.4.1 Incidence Matrix

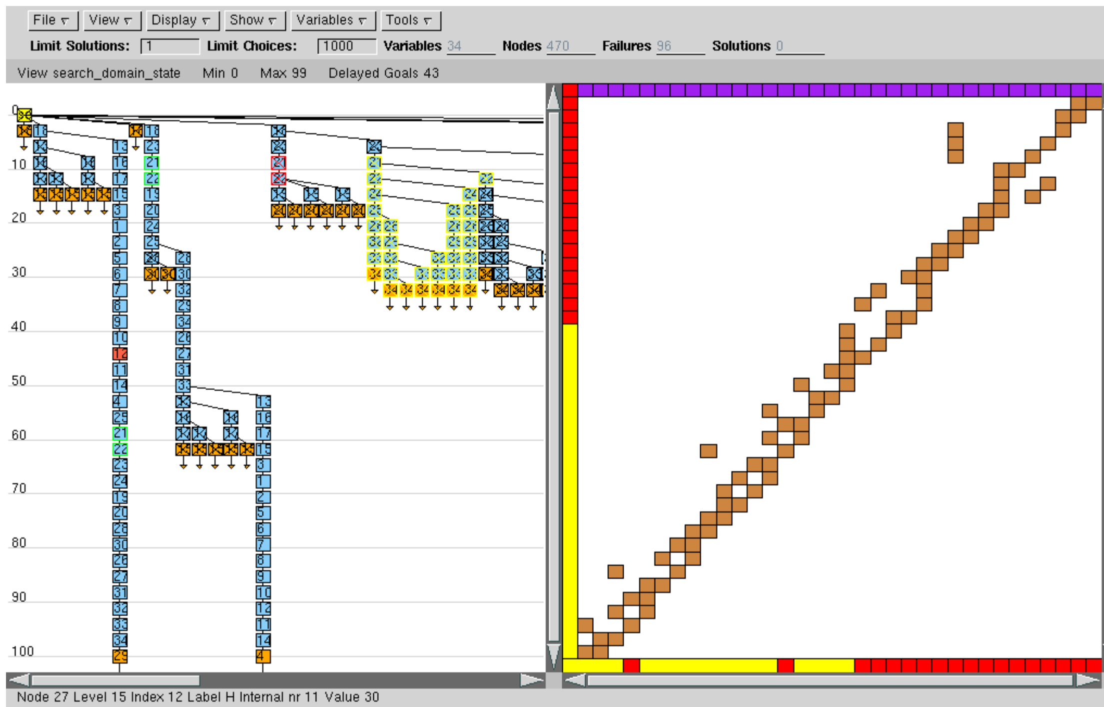


Figure 5: Constraint Incidence matrix

All constraints can be shown as a constraint variable incidence matrix. 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 6 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.

1.5.4.2 Update View

This view shows the activity of the constraints in the different levels 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 update (min, max, remove)

– the constraint binds some value

– the constraint is solved

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.

1.5.4.3 Constraint Count

This simple view shows a graph 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 [AB93] [BC94]. 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 will be described in a forthcoming paper.

1.5.5 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 7 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. 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.

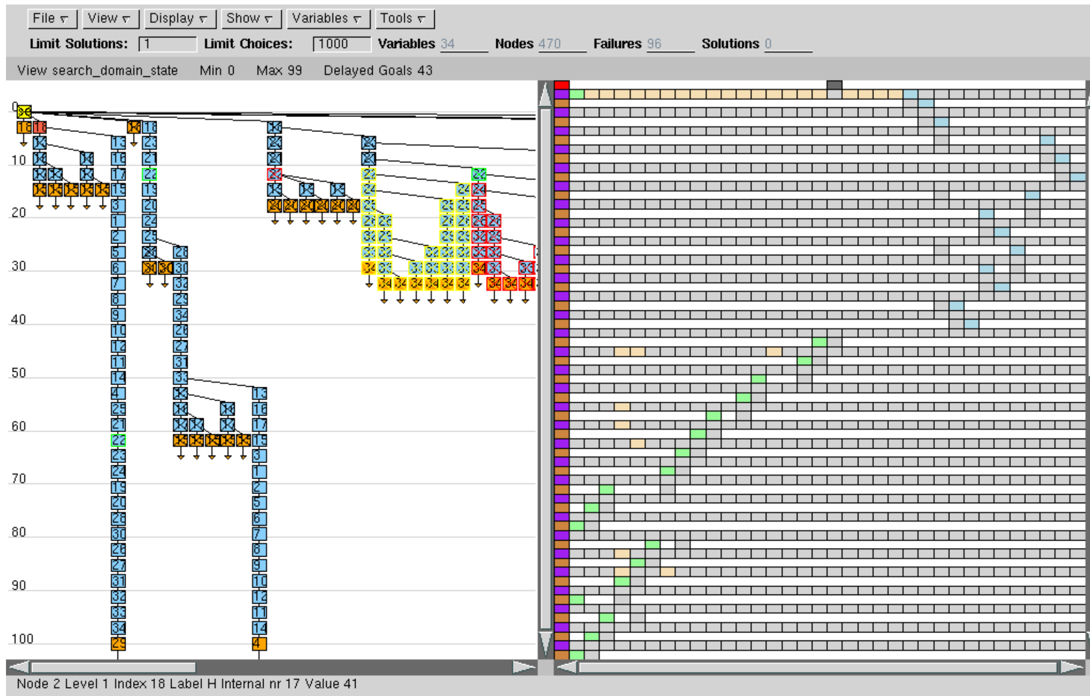


Figure 6: Propagation view

2 N-Queens

2.1 Problem

The problem is to place N queens on a chessboard of size NxN in such a way that they do not attack each other. The initial problem is stated for the 8x8 board, but it is more interesting to consider all sizes from 4 to 200, say. Note that this problem is not hard, in fact you can construct generic solutions for different board size pattern. But the N-queens problem is considered a classical demo for AI (artificial intelligence) programming and it is often used as a benchmark problem to compare different implementations of constraint systems.

2.2 Model

There are many different possible models for the N-queens problem, but two of them are regularly used for constraint systems. As we have to place N non-attacking queens on the NxN board, we immediately see that each column of the board must contain one and only one queen. By identifying the row in which the queen is placed, we uniquely describe a placement of the queens on the board. In constraint terms, we use N variables with a domain of 1..N each.

To express the constraints, we have two alternatives.

2.2.1 Binary Constraints

The first variant considers constraints between pairs of variables. For two queens placed in columns i and j respectively, the following constraints must hold:

$X_i \backslash= X_j$

$X_i \backslash= X_j + j-i$

$X_i \backslash= X_j + i-j$

The first constraint states that queens are not in the same row, the second and third constraints state that they are not on the same diagonal. We have to create these constraints for all pairs of variables, so we need a quadratic (in the number of variables) number of constraints.

2.2.2 Alldifferent constraints

The second model drastically reduces the number of constraints required. It is due to Y.C. Cras, who first presented this model for Charme. Instead of expressing binary constraints, we use three alldifferent constraints. The first constraints states that all queens must be placed in a different row, i.e.

```
alldifferent([X1,X2,..,Xn]),
```

The two other constraints state that all queens must be placed on different diagonals. This is expressed by adding the column number to the variables

```
alldifferent([X1+1,X2+2,...,Xn+N]),

alldifferent([X1+N,X2+N-1,...,Xn+1]),
```

This model has the advantage that only three constraints are needed in the model regardless of board size. If a syntactic version of the alldifferent constraint is used, exactly the same propagation is used as for the binary constraints.

2.2.3 Search Method

The classical assignment method for the N-queens problem uses the first-fail strategy. This strategy selects at each step the unassigned variable with the smallest domain. If several variables have the same domain size, then some tie break method must be used. This selection can be done in different ways. Our example below shows the impact of this tie-break method.

2.3 Program

The program below uses the alldifferent constraint model and the first-fail selection strategy. The predicate *top/0* calls the problem solver for a board size of 40, the predicate *solve/1* contains the actual problem solver. The lists *L1* and *L2* contain terms of the form X_i+i or $X_i+N-(i-1)$, the list *K* contains terms of the form $t(X_i, i)$.

```
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),
        label(K).

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),
        indomain(X),
        label(Rest).
```

Below we show the interface to the search tree tool. We load the library search and use the predicates *search_start/3* and *search_node/3*.

In the first argument of *search_start/3* we pass the list of variables to be assigned, in the second argument the call to the search routine and in the third argument some options for the search tree tool, here the width and height of the search tree tool main window.

The call to the choice predicate *indomain/1* is wrapped in a call to *search_node/3*. The first argument is the variable that is assigned, the second argument its index in the list of variables and the third argument is the choice point generating call.

```
?-lib search.
```

```
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),
        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).
```

2.4 Solution

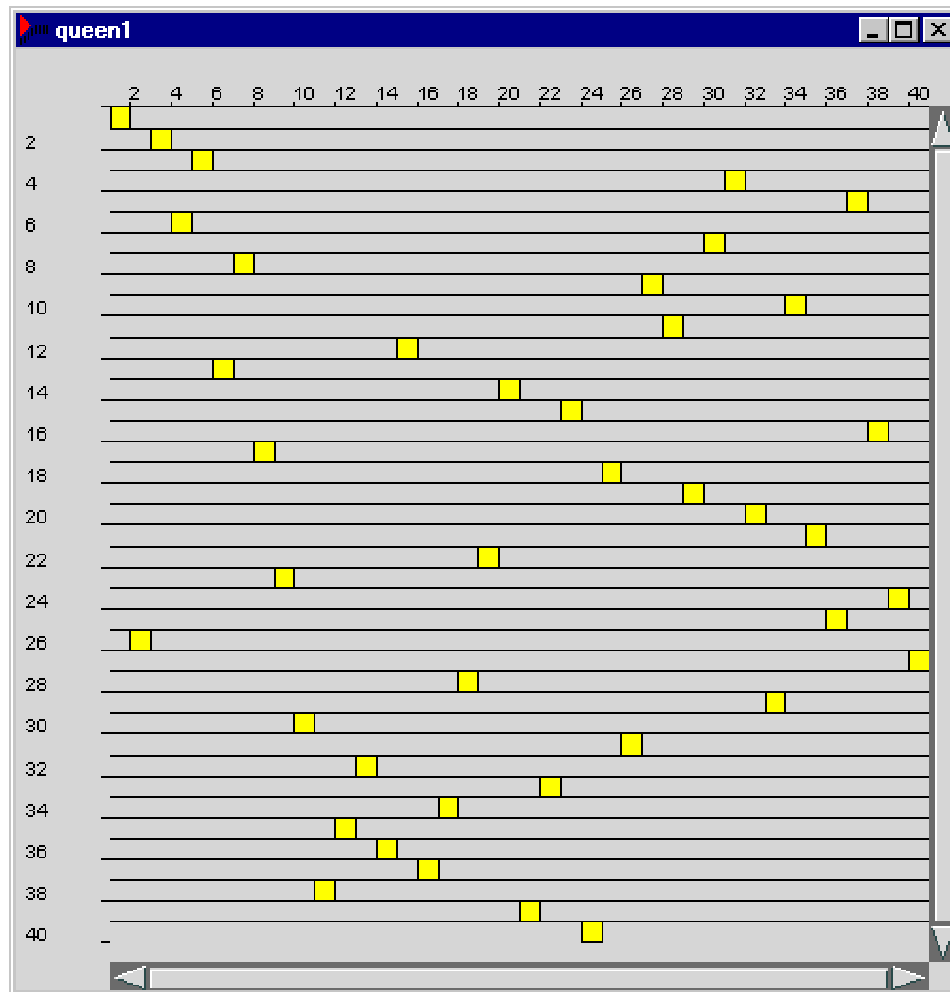


Figure 7: Solution for 40-queens

Just looking at the result does not tell us much. A look at the searchtree and the state of the computations contains a lot more information.

2.5 Searchtree

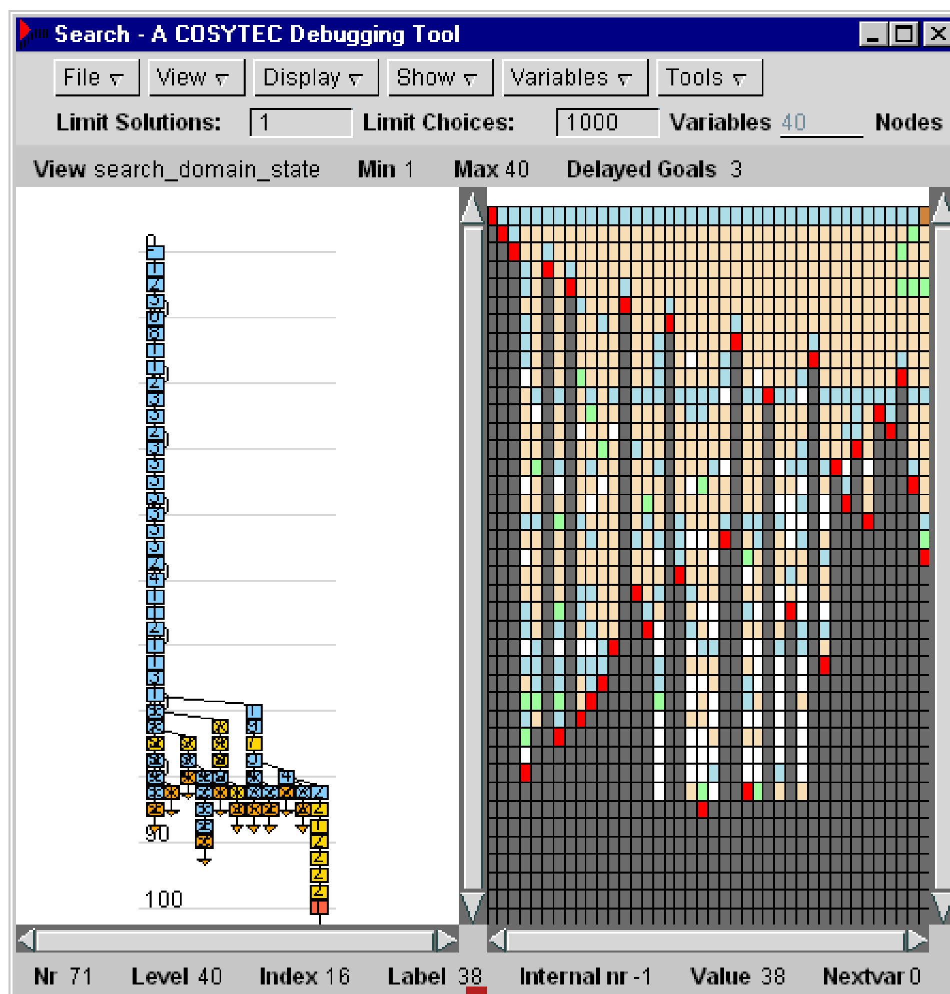


Figure 8: Searchtree for 40-queens

We can observe the following facts:

- The smallest values are taken quite early, at the end only the high values are left in the domain.
- The variables are selected in a rather dynamic way, not strictly left to right. At the end, the last remaining variables are in the center of the board.
- At every step, some values are removed from most variables. This means that we have to remember all these changes, so that memory for trailing will be typically quadratic in the number of variables.
- Failure in the tree is normally caused by a set of few remaining variables for which not enough compatible values exist.

2.6 Analysis

So far, so good! We can find the solution for the 40 queen problems in a reasonable amount

of time with just a few backtracking steps. But if we compare results of other systems, we can get a surprise. As an example take the 96 queens problem solved with forward checking and a first fail selection method. Different systems return quite different results:

System Backtracking	Steps
CHIP delete	28
CHIP labeling	558
CLP(FD)	45
B-Prolog	49
Sicstus	???

Thanks to N.F. Zhou and M. Carlsson for the results in the comparison.

How can be explain these differences? As all systems use the same propagation method and the same selection strategy, the difference must lie in the tie-break selection

We will study this problem with two variants of the same CHIP program, one using *delete/5* as shown above, the other using *labeling/4*.

2.7 Comparison

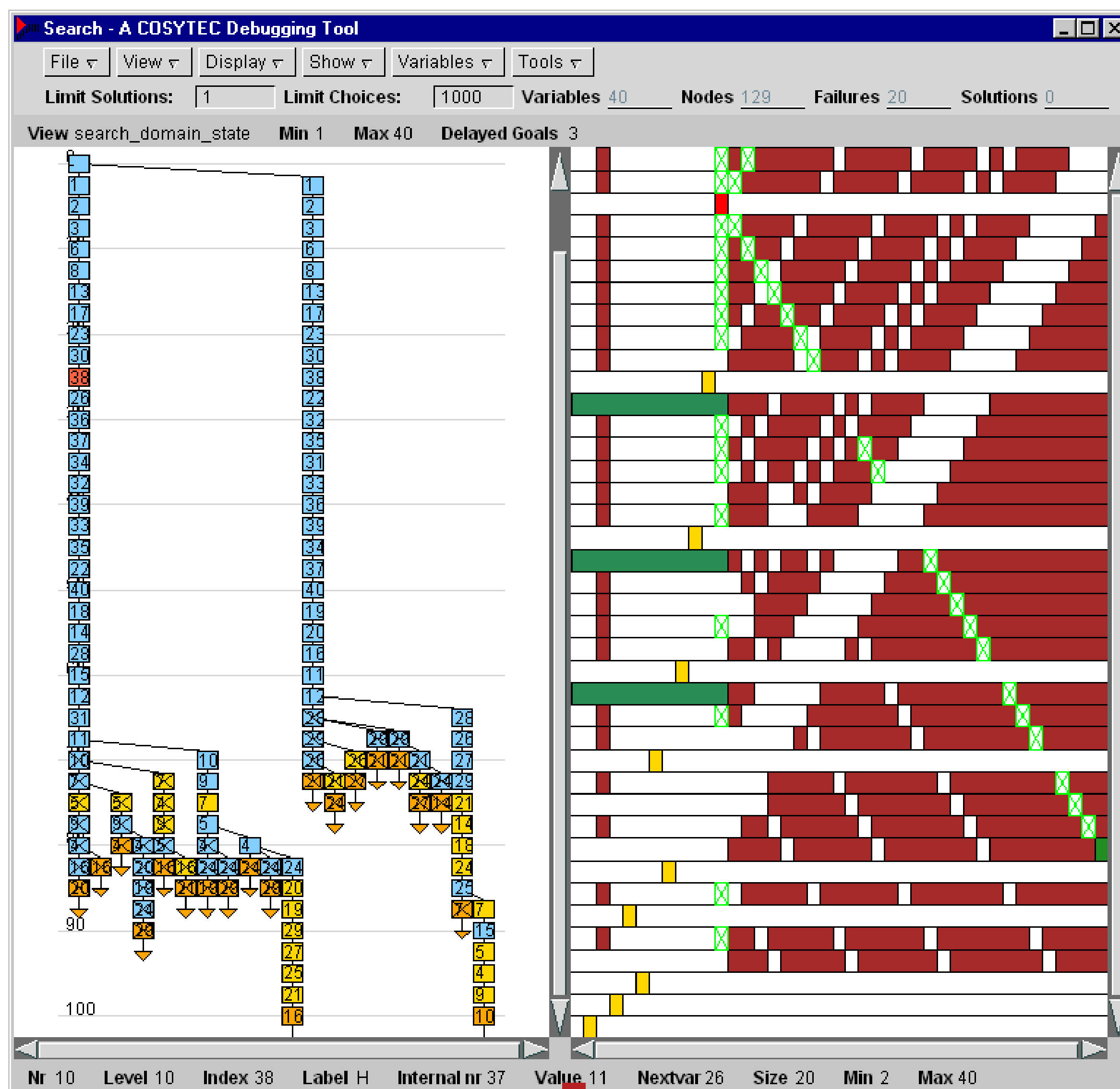


Figure 9: Comparison of two search methods

In this version of the program we use an alternative clause in the labeling to choose among two different methods. After finding the first solution, we backtrack to this disjunction and try the other branch. In the display we see basically two search trees side by side.

At first glance, the two trees are not so different. After assigning 60-70% of the variables, we start backtracking and eventually find a solution.

Looking more closely, we see that the trees are quite different. In the picture we have selected the node with index 38. Up to this point the trees are the same, after that the sequence of the selected variables starts to change.

2.8 Phaseline

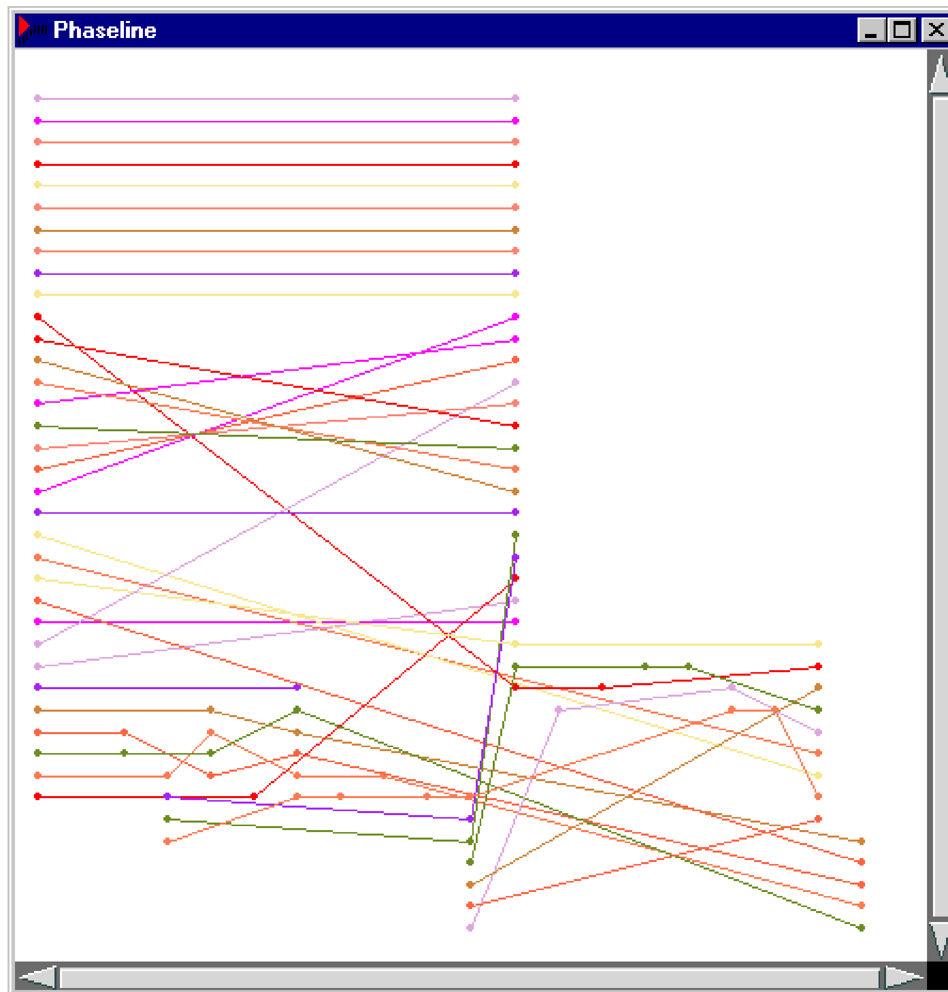


Figure 10: Phaseline display

The phaseline display clearly shows the difference in the search procedures. Lines connect nodes in the searchtree which assign the same variable. At the top of the tree, the lines are parallel; this means that the same variables are assigned at each level. We can then see major differences in the variable selection, both strategies enumerate the variables in a very different order.

2.9 Overlay

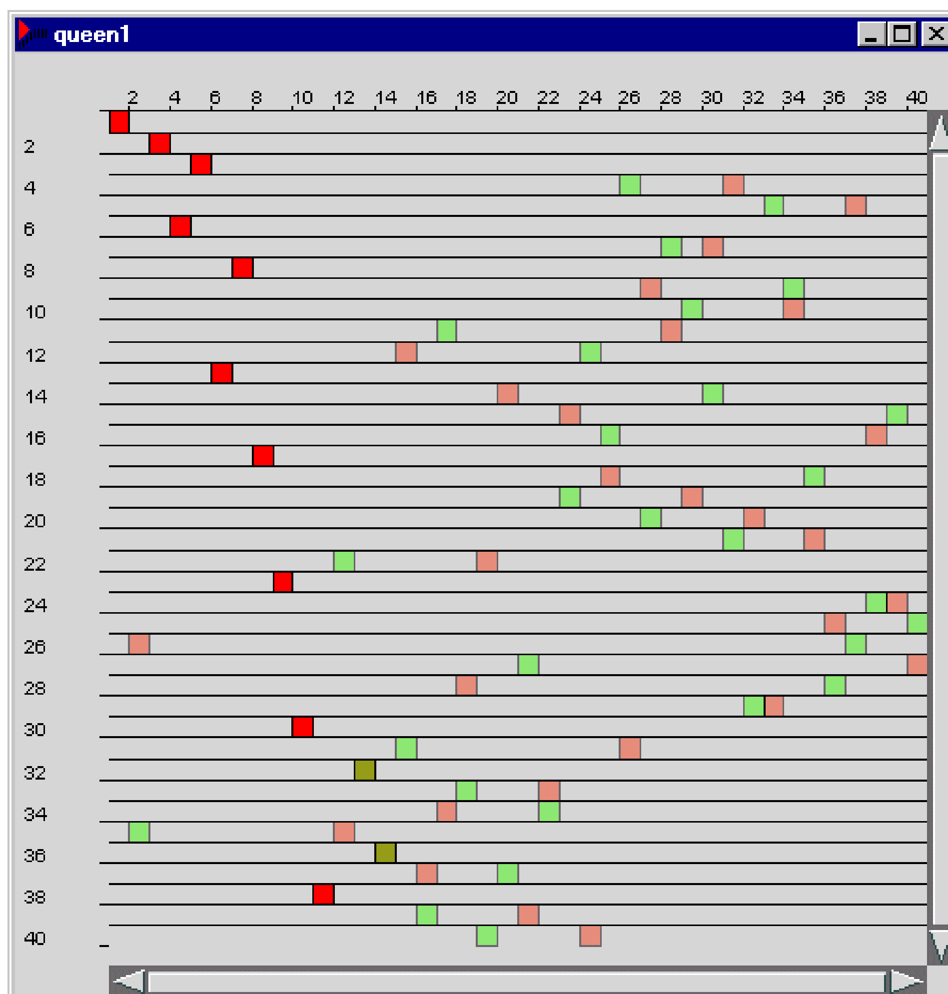


Figure 11: Solution Overlay

But does the order matter? The picture above shows an overlay of the two solutions found. Values in a bright red have been assigned in the common part of the trees. The pale red and green values are assigned in the two different trees. The brighter olive color shows the two instances where some variable is assigned to the same value in both solutions. Clearly, the solutions are quite different. Starting with a single variable selection at the top, the solutions diverge very rapidly.

The visualization tools are instrumental in discovering these differences. It would be tedious at least, probably even difficult to discover and explain such differences without redeveloping some visualization tools.

2.10 Variable Selection

We have discussed the difference between the selection mechanism in delete/5 and in labeling/4. So, which of the methods is better? To evaluate this, we run the problem on all sizes from 4 to 200 and count the backtracking steps. Figure 1 shows the number of backtracking steps for delete/5, figure 2 the backtracking steps with labeling/4.

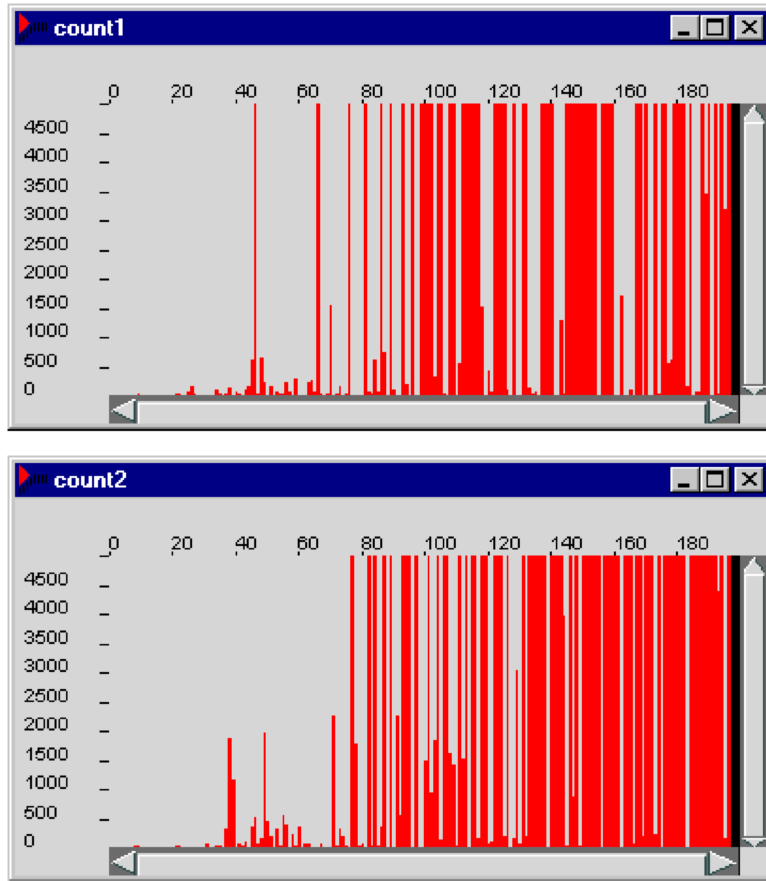
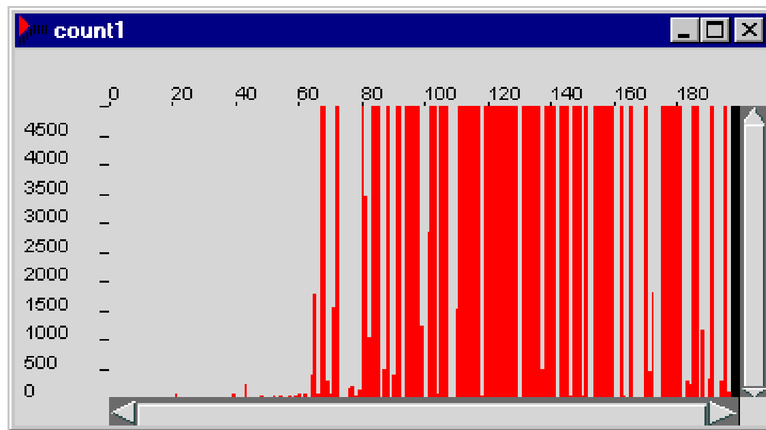


Figure 12: Solution for 4-200 queens

We stop the search after 5000 backtracking steps. In the picture we can se that we hit the limit quite often, especially with board sizes above 80. One interesting observation is that quite often one of the methods finds a solution, but not the other, but neither method is really successful.

We can now try another selection mechanism, starting from the center of the board. We still use the first-fail strategy, but reorder the variables statically so that the variables in the middle are placed first.



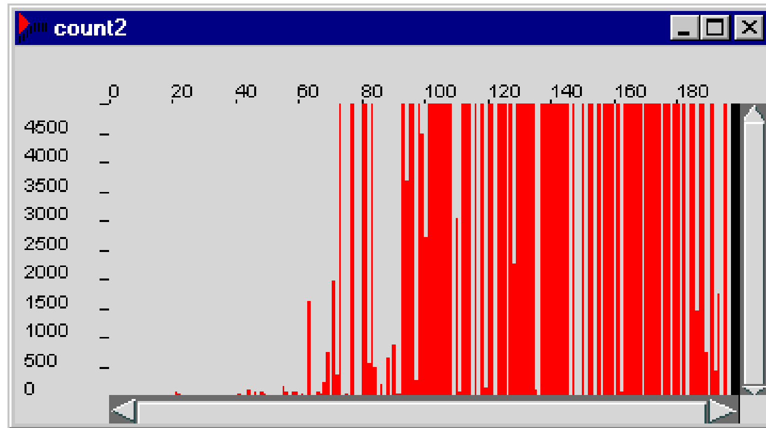


Figure 13: Solutions for different selection method

Again, it is not clear which method is better. But we will use this center-based selection method together with a value choice method with very good results.

2.11 Value Choice

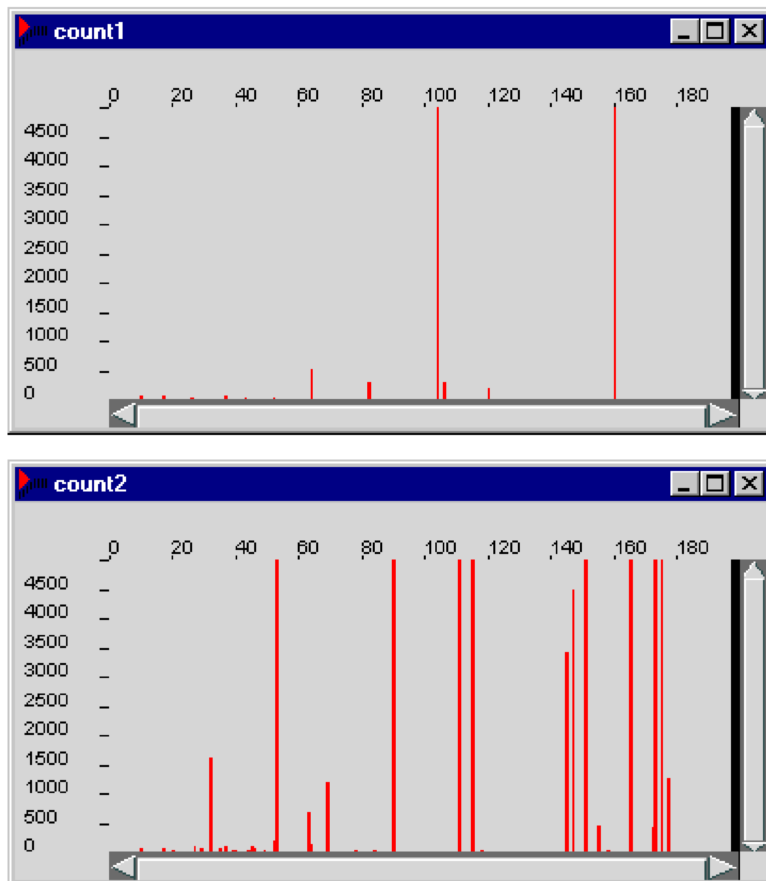


Figure 14: Solution for different choice method

In the previous version, we have used the standard indomain/1 predicate to assign values. This is quite a good starting point, but we can think of some improvement here. Initially, we

should select the values which are most difficult to assign. Starting the assignment from the center of the board is a good approximation of this strategy.

We look at two program variants here, one using delete, the other labeling/4. We can see in the diagrams that we obtain much better results, although we still do not find a solution for some problem instances. The version with delete is clearly superior in that respect, it also typically uses less backtracking steps to find the solution.

2.12 Credit based search

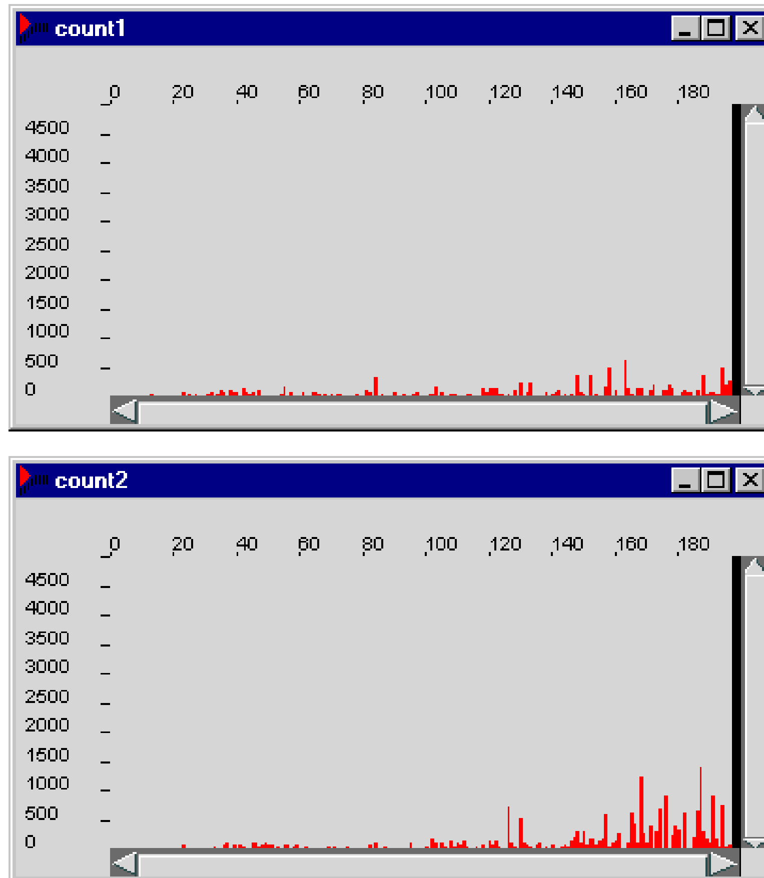


Figure 15: Solution for credit based search

An alternative is to use credit based search. This partial search technique stops the enumeration after a few steps and jumps back to the top of the search tree to try an alternative.

Unfortunately, the diagram above paints a rather optimistic picture. Both selection methods find solutions for all problems with rather few backtracking steps, but it still takes quite some time to find the solutions. An even better solution is obtained by combining variable selection, value selection and partial search together.

2.13 Combination

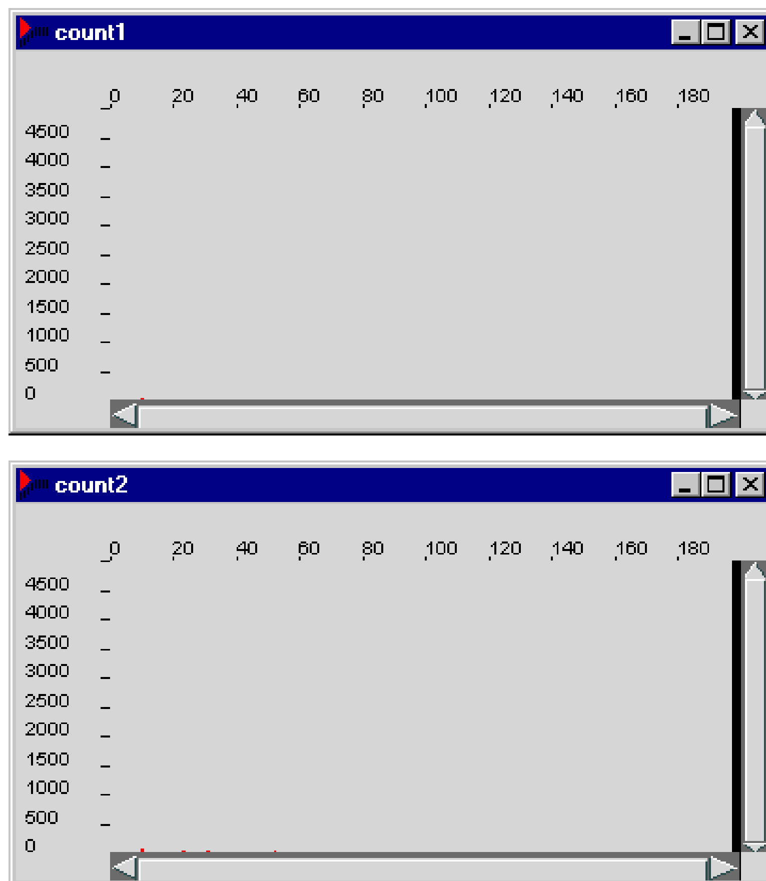


Figure 16: Solution for combination of methods

If we combine all techniques, we come close to the ideal program. For all problem instances, we only use a few backtracking steps to find a solution. There are no problems with "particularly hard problems", since we use credit based search to jump back to a higher level as soon as more than a few backtracking steps occur. As the problem is not really hard, we find a solution quite quickly.

2.14 Summary

2.14.1 Simple Problem

- 3 constraints for any problem size
- possible to construct solutions

2.14.2 Small changes have big impact

- variable selection tie break
- very different solution
- incomparable results

2.14.3 Improvements

- value + variable selection from center
- most constraining
- partial search
- best result by combination

3 Map Coloring

3.1 Problem

This is another well-known constraint demo, originating from a mathematical puzzle column of M. Gardner in the Scientific American. The problem consists in coloring a map with 110 countries with four colors in such a way that any two neighboring countries do not have the same color. The map is shown below:

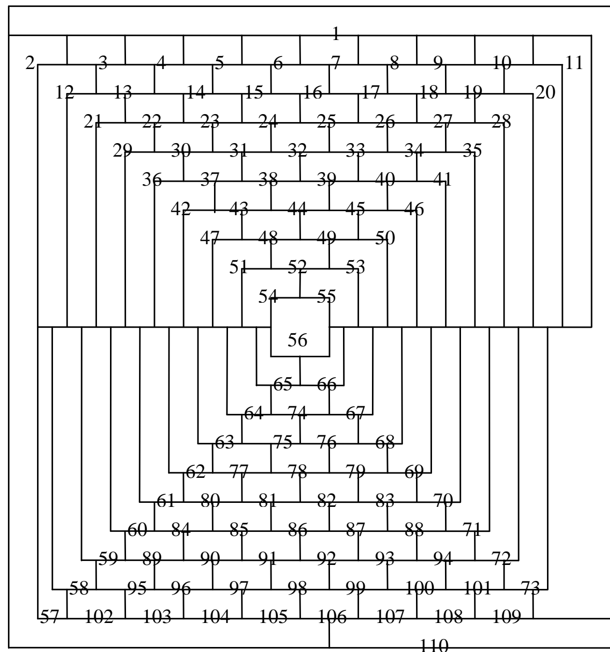


Figure 17: Map coloring problem

From the four color theorem it is clear that this problem has a solution, but finding a solution by hand is quite time consuming. Solving the problem with CHIP turns out to be quite simple.

3.2 Model

The model of this problem is straightforward. Each country is represented by a domain variable which ranges over four possible values. We state a disequality constraints between all pairs of variables which represent neighboring countries and search for an assignment of values which satisfies all constraints. This way the constraint of the problem will be satisfied.

3.3 Program

The program below uses CHIP++ objects to represent the countries. Each object has a domain variable color, on which the disequality constraints will be expressed. The connectivity of the map is expressed with a binary predicate n/2, which lists the neighboring countries. This form of program is preferable to a "open-coded" version where the constraints are expressed explicitly between 110 domain variables, as we clearly separate data from the constraint model.

Note that we have included the meta-predicates for the search tree generation.

```
?-lib search.

?-not class(country), create_class(country,object,[color]).

top:-
        countries(C),
        create_objects(C,Vars),
        findall(n(A,B),n(A,B),L),
        setup(L),
        search_number(Vars,K),
        search_start(Vars,label(K),[winw<-760,winh<-500,choices<-20000]),
        writeln(Vars).

label([]).
label([H|T]):-
        delete(t(X,N),[H|T],R,1,input_order),
        search_node(X,N,indomain(X)),
        label(R).

create_objects([],[]).
create_objects([C|C1],[V|V1]):-
        create_country(C),
        V :: 1..4,
        C@color = V,
        create_objects(C1,V1).

create_country(C):-
        instance(C),
        !.
create_country(C):-
        create_instance(C,country).

setup([]).
setup([n(A,B)|N1]):-
        A@color #\= B@color,
        setup(N1).

/*

DATA

*/

countries([c1,c2,c3,c4,c5,c6,c7,c8,c9,
c10,c11,c12,c13,c14,c15,c16,c17,c18,c19,
c20,c21,c22,c23,c24,c25,c26,c27,c28,c29,
c30,c31,c32,c33,c34,c35,c36,c37,c38,c39,
c40,c41,c42,c43,c44,c45,c46,c47,c48,c49,
c50,c51,c52,c53,c54,c55,c56,c57,c58,c59,
c60,c61,c62,c63,c64,c65,c66,c67,c68,c69,
c70,c71,c72,c73,c74,c75,c76,c77,c78,c79,
c80,c81,c82,c83,c84,c85,c86,c87,c88,c89,
c90,c91,c92,c93,c94,c95,c96,c97,c98,c99,
c100,c101,c102,c103,c104,c105,c106,c107,c108,c109,
c110]).


n(c2,c1).
```

```
n(c3,c1).
…
n(c110,c109).


?-top.
```

3.4 Solution

A graphical representation of the solution is given below:

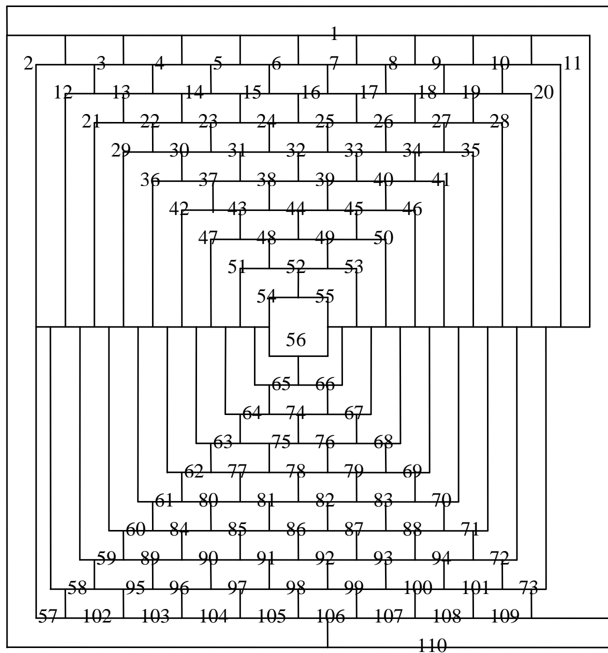


Figure 18: Map coloring solution

We at the moment do not have a visualizer to represent this type of problem, the graphical output was generated with a custom program.

3.5 Searchtree

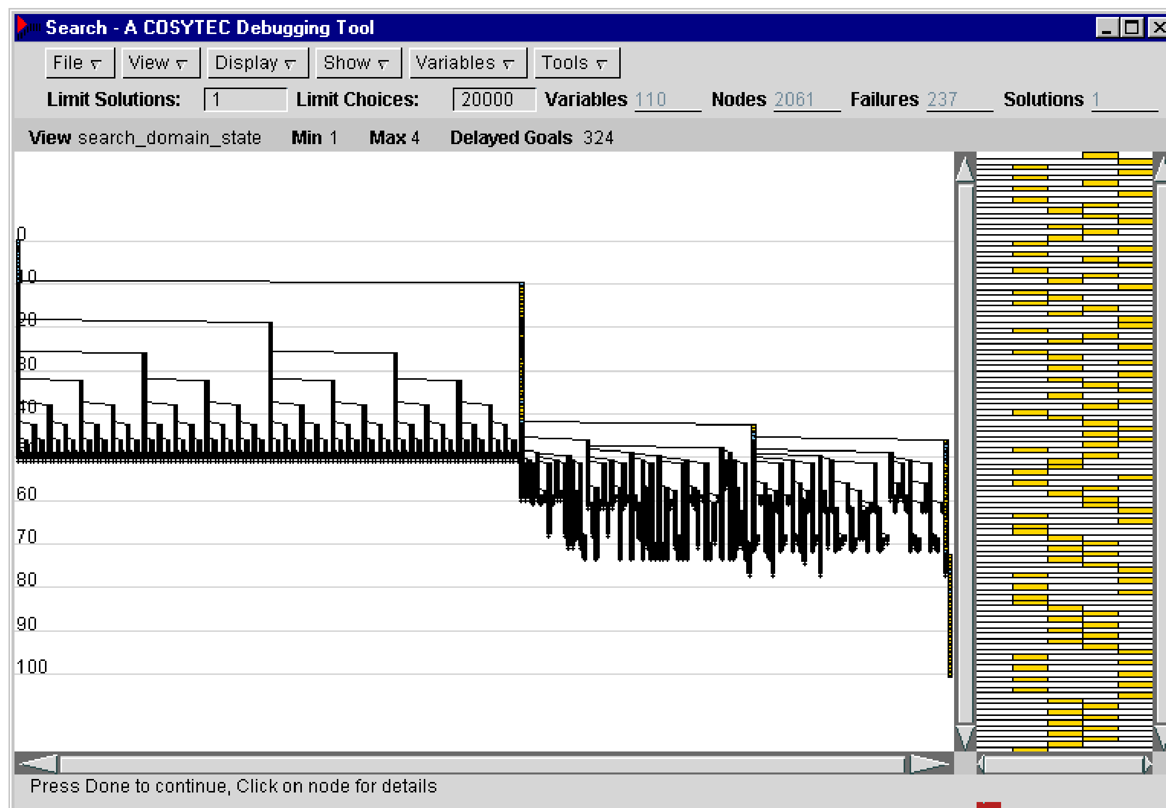


Figure 19: Searchtree for map coloring

The searchtree to find the solution is quite big, just over 2000 nodes. We can distinguish on the left part of the tree a very regular structure, where the search routine is thrashing. Eventually we backtrack high enough in the tree to escape this problem, but then still have to explore a large number of nodes before finding a solution.

In domain state view we see the values assigned to each variable. As we are using the input_order, the variables are assigned from bottom to top. In the initial part of the assignment we see a rather regular structure. Certain values are clearly preferred in certain parts of the tree.

3.6 Analysis

3.6.1 Slow to find first solution

The difficulty in finding a solution and the trashing points at a problem with the constraint propagation and the variable selection.

3.6.2 Value choice too predictable

The indomain predicate used to assign values always will choose small values before the larger ones. This means that the value four in this example will only be used when no other alternative exists. This imbalance can create problems if the constraint propagation does not foresee potential difficulties.

3.7 Domain Update

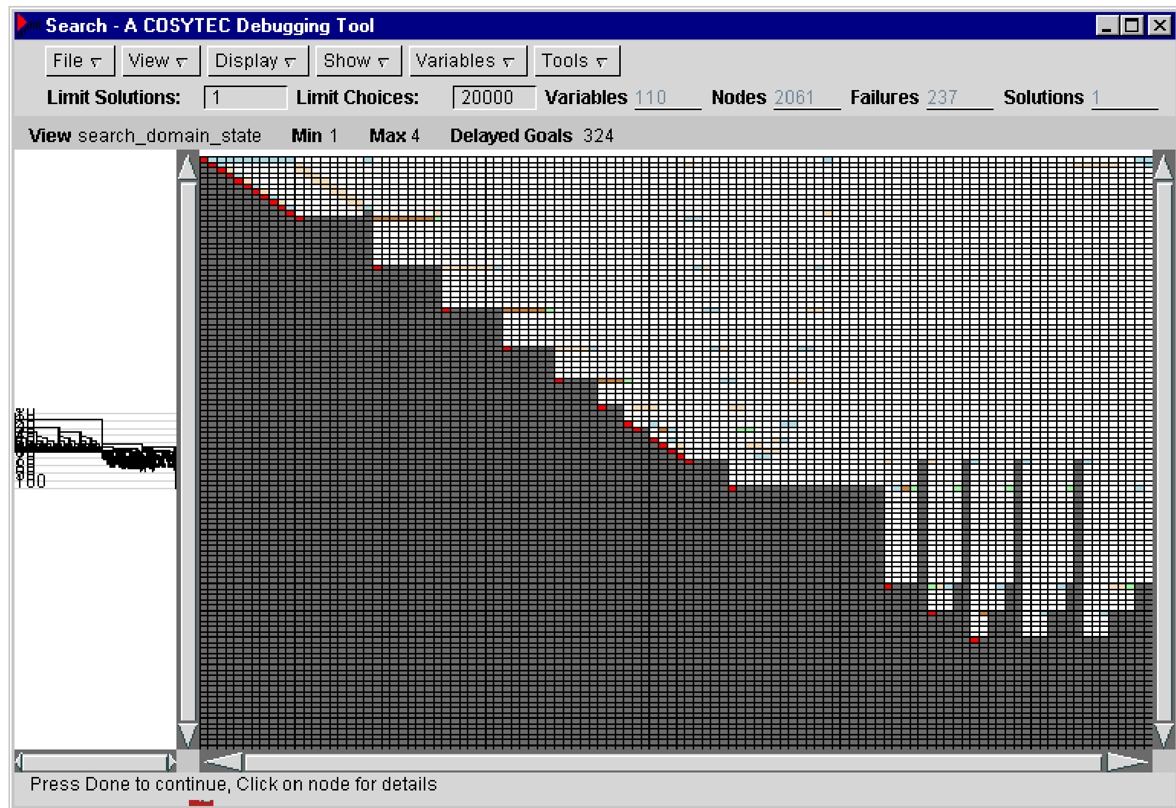


Figure 20: Domain update view

This picture shows the domain updates on the path to the first solution. We can see the variable selection following the input order, with a significant number of variables assigned by propagation. We can also remark that there is not very much domain reduction by propagation. This is partly due to the constraints used, partly due to the small domains of the variables.

3.8 Incidence Matrix

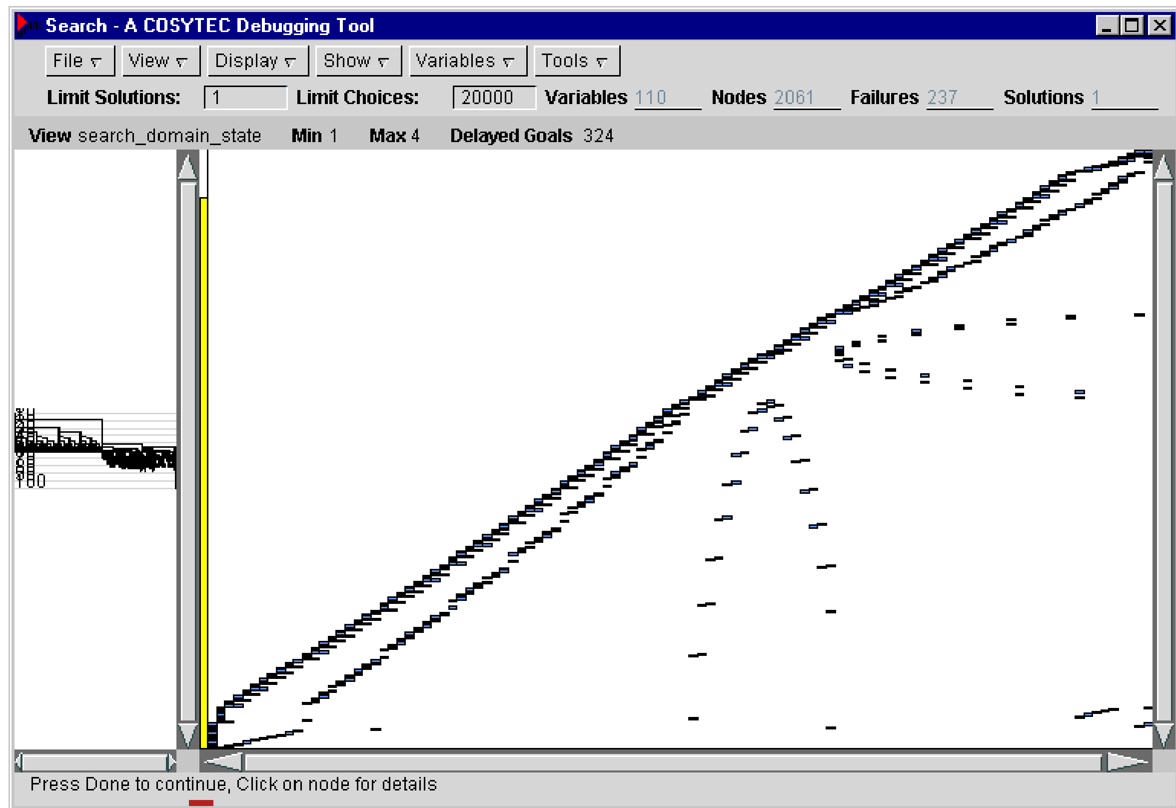


Figure 21: Incidence matrix

Just to show that there is a lot of hidden structure in the relation of country names and constraints, we can look at the incidence matrix of the problem. While the exact relation between the countries is not immediately visible, it is clear that there is a structural relation.

For comparison, here we have the incidence matrix if we re-order the variables in a random way:

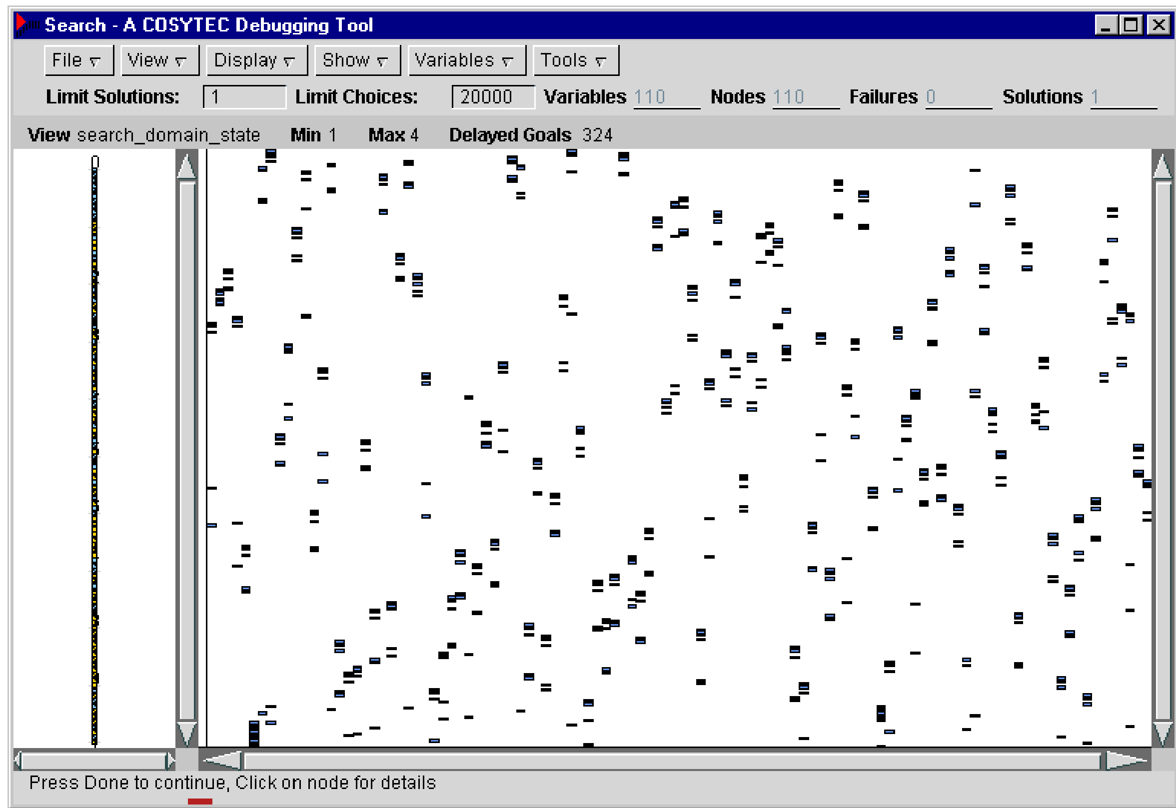


Figure 22: Incidence matrix with random order

3.9 Selection

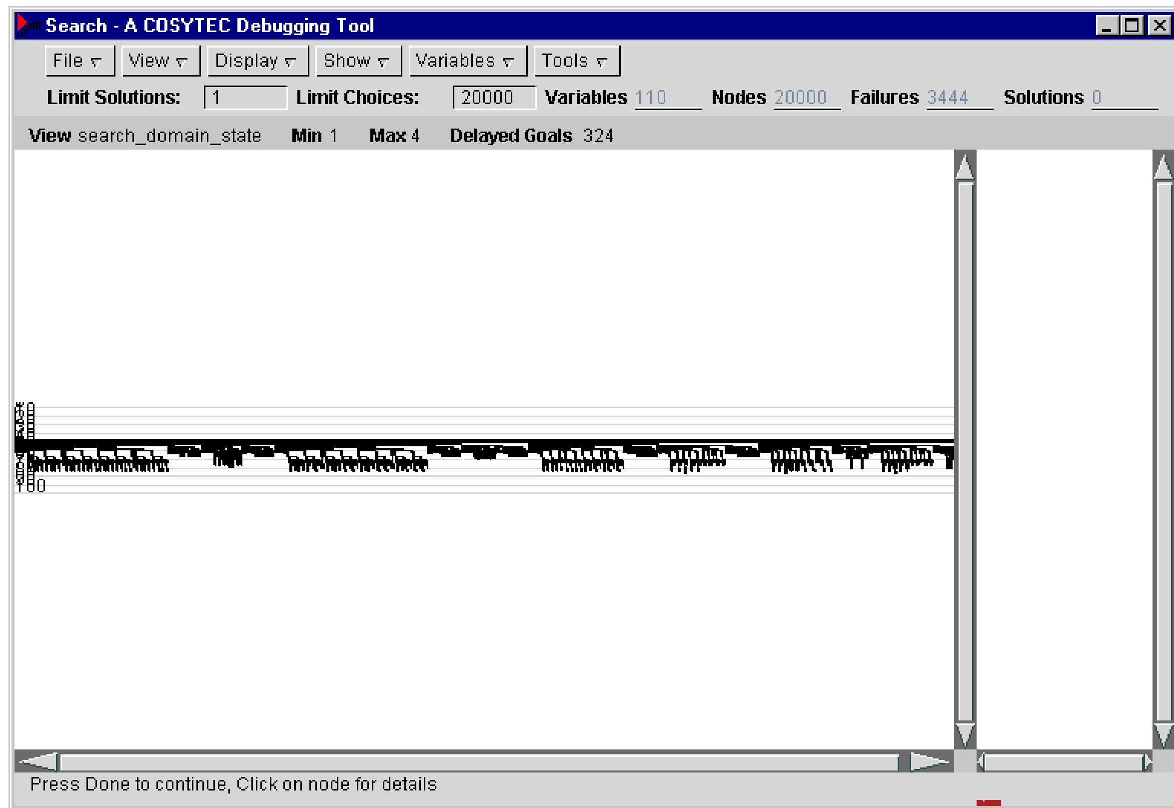


Figure 23: New variable selection

The first idea to improve the solution is to use a dynamic variable ordering using the delete/5 predicate.

For this type of problem with a small domain size and a uneven distribution of constraints between variables, the most_constrained strategy looks the most promising. The picture above shows the result. Within 20000 nodes, the system does not find a solution at all! Even without looking at the tree in detail, we can recognize regular structures where the search is thrashing.

What is happening here? Do variable orderings not work as well as we thought?

3.10 Selection

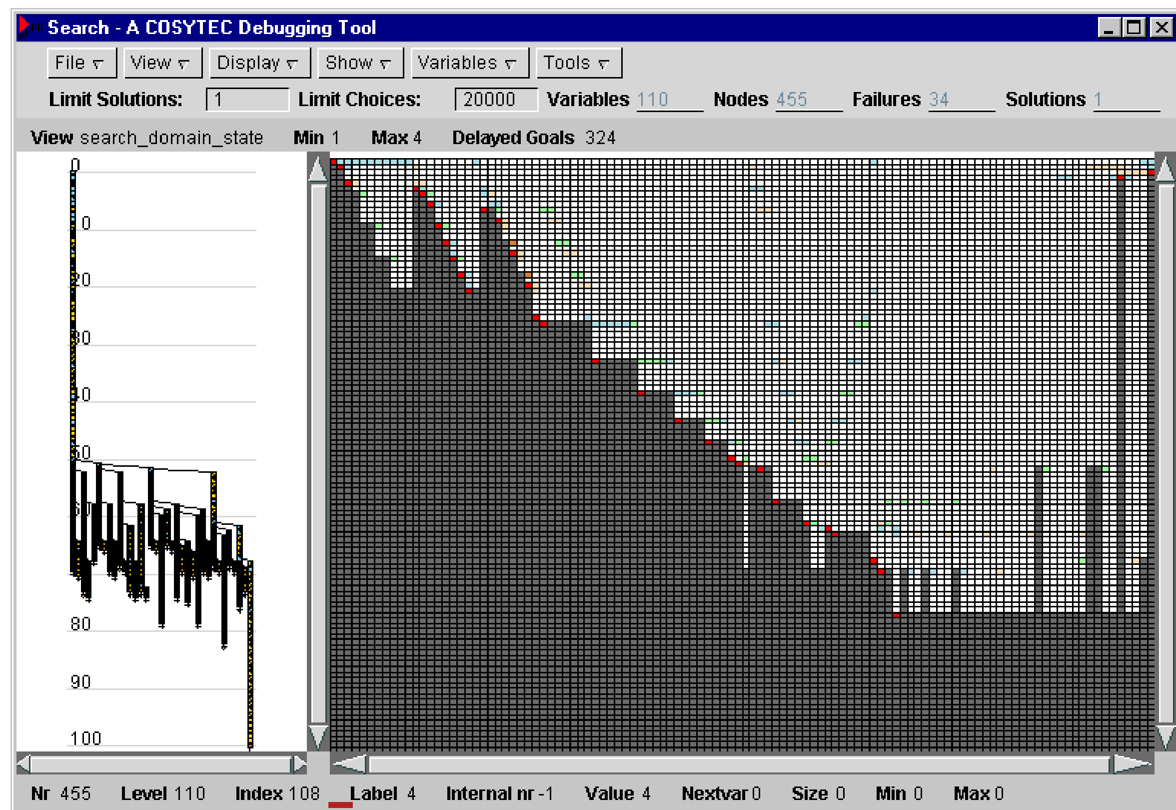


Figure 24: Another variable selection

This again is the most_constrained selection strategy, but this time the one of labeling/4. What a difference, a solution is found with 455 nodes! Clearly, the tie breaking method has a big impact in this problem. The delete method reorganizes the list of variables at each step, so that the natural order of the variables is not preserved. The labeling selection method keeps the natural order, which in this case is much preferable.

3.11 Value Choice

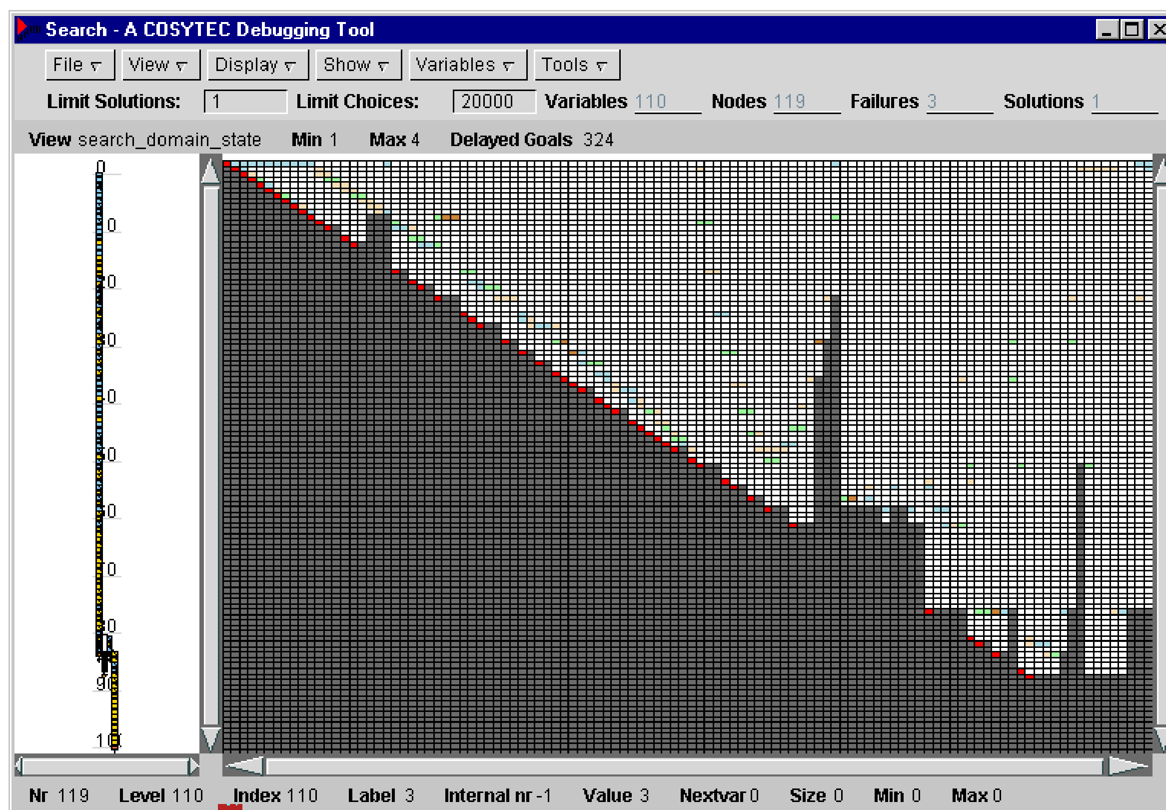


Figure 25: Value choice

This picture shows the search tree if we change the value selection. We noticed above that indomain uses small values in preference to large ones, which creates an uneven use of these values. We have modified the program to more evenly assign colors by rotating the initial color to be used. The simple program is shown below (as usual, only the modified part is shown):

```
top:-
        countries(C),
        create_objects(C,Vars),
        findall(n(A,B),n(A,B),L),
        setup(L),
        search_number(Vars,K),
        search_start(Vars,label(K,[1,2,3,4]),[winw<-760,winh<-500,choices<-
20000]),
        writeln(Vars).

label([],_).
label([H|T],L):-
        delete(t(X,N),[H|T],R,1,input_order),
        search_node(X,N,member(X,L)),
        rotate(L,K),
        label(R,K).

rotate([A,B,C,D],[B,C,D,A]).
```

This simple modification dramatically reduces the search space, there are just three failures in

the tree.

3.12 Combination

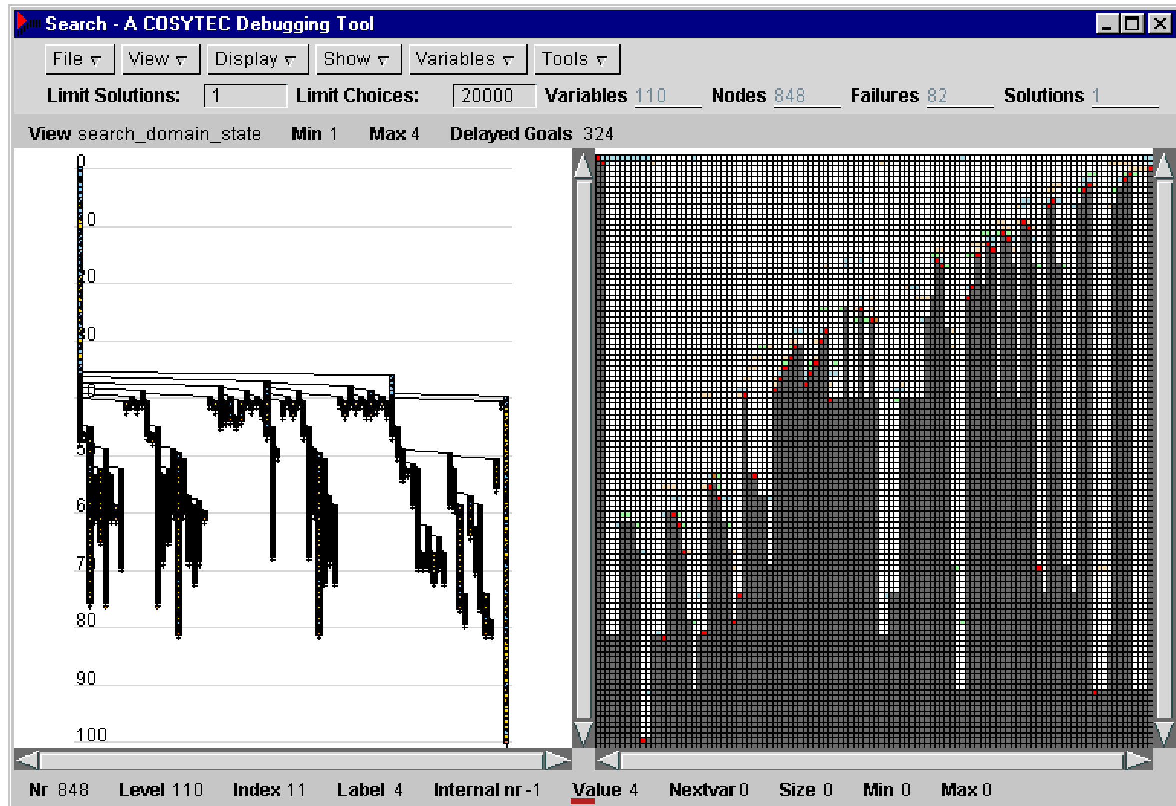


Figure 26: Combination of technique

What happens if we combine the two techniques? If we use the delete variable selection, we eventually find a solution, after creating 848 nodes. Looking at the sequence of the variable selections in the right-hand diagram, we see that the order is nearly static, but runs from the right to the left, quite the opposite of the input_order.

3.13 Combination

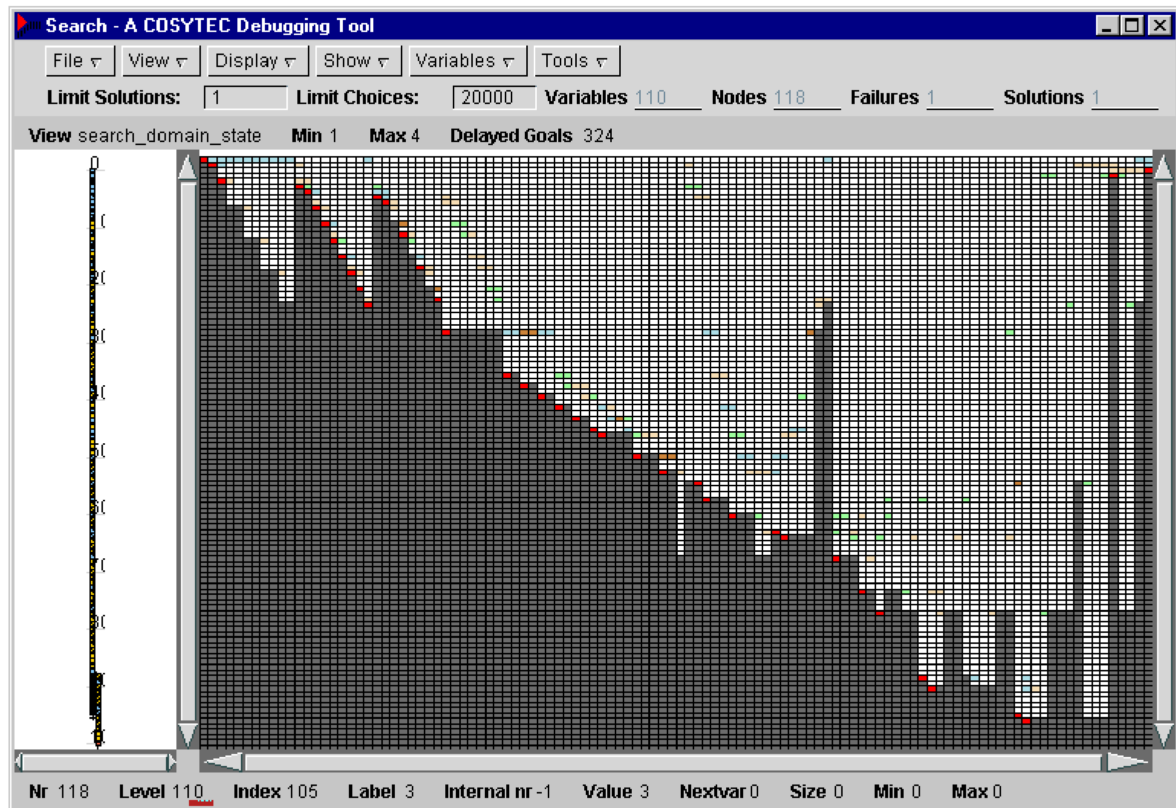


Figure 27: Combination with labeling

Using the labeling selection with most_constrained and the rotating color assignment, we obtain a nearly perfect search tree. A single failure is detected before the solutions is found. The variable selection is mostly left to right, with the notable exception of countries 110 and 106, which are assigned very early on.

We can also see that there is not very much propagation, most of the fields are not colored, but either white (unchanged) or dark gray (assigned). We can also see that quite a few values are fixed by propagation, when a constraint removes one of the remaining two colors for a country.

3.14 Redundant Constraints

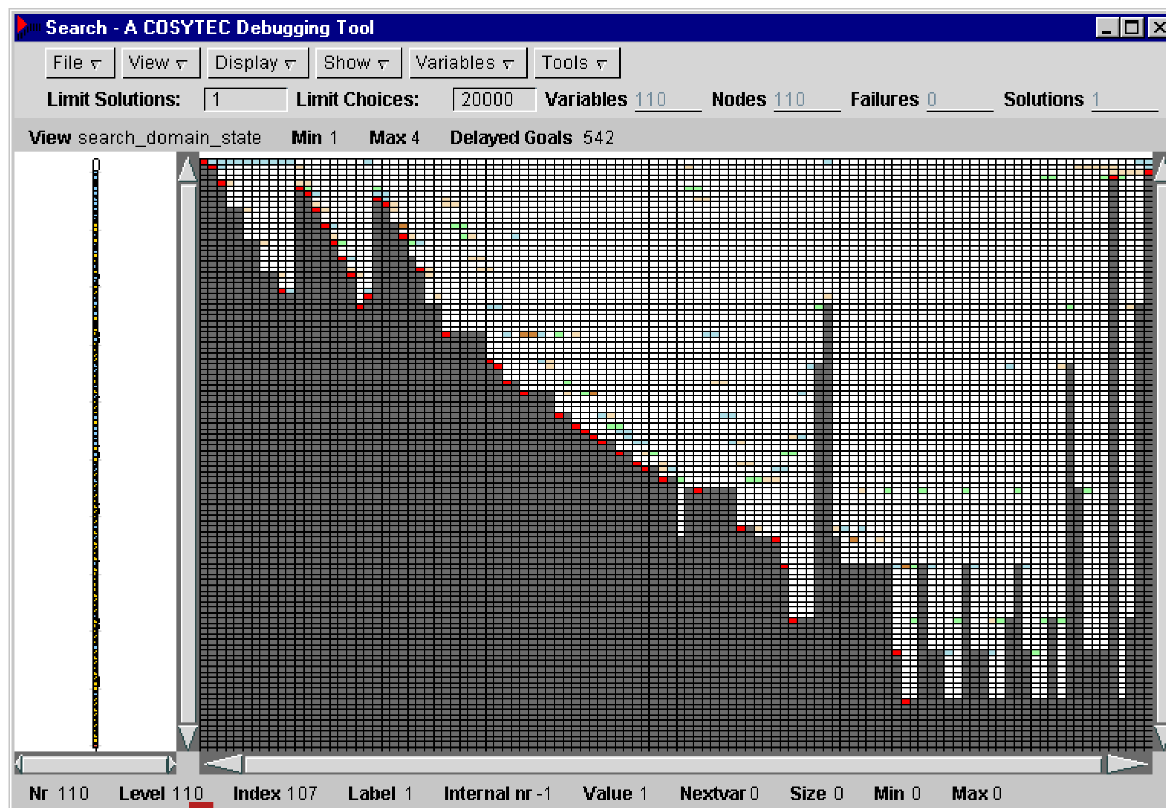


Figure 28: Redundant Constraints

There is no real need to further improve the program, but just for completeness we try to add some redundant constraints. Clearly, the forward checking used by the disequality constraint is not powerful, and we know much better constraint reasoning methods for the alldifferent constraint. We can use these techniques by generating all cliques of three or four nodes in the graph and generating redundant constraints for them. As the alldifferent constraint in CHIP only uses forward checking, we use the diffn constraint, which contains a perfect matching method to deduce failures more rapidly.

We can create all cliques in this graph with a little Prolog program:

```
set(A,B,C):-
        n(A,B),
        n(B,C),
        n(A,C).

set(A,B,C,D):-
        n(A,B),
        n(B,C),
        n(C,D),
        n(A,C),
        n(A,D),
        n(B,D).

redundant(_):-
        findall(set(A,B,C),set(A,B,C),L),
        redundant_lp(L),
```

```
        findall(set(R,S,T,U),set(R,S,T,U),K),
        redundant_lp(K).

redundant_lp([]).
redundant_lp([set(A,B,C)|R]):-
        diffn([[1,A@color,1,1],[1,B@color,1,1],[1,C@color,1,1]]),
        redundant_lp(R).
redundant_lp([set(A,B,C,D)|R]):-
        diffn([[A@color,1],[B@color,1],[C@color,1],[D@color,1]]),
        redundant_lp(R).
```

We now find the solution without any backtracking! But is this again due to some small change in the variable selection, or to improved constraint reasoning? We can answer this by comparing the domain update diagrams for the two programs with a graphical overlay. The dark colors are parts common to both programs, the light gray areas show variables which are assigned earlier in the redundant program. We can also see (by following the red assignment markers) that the variable selection is not quite the same, but still very close to each other.

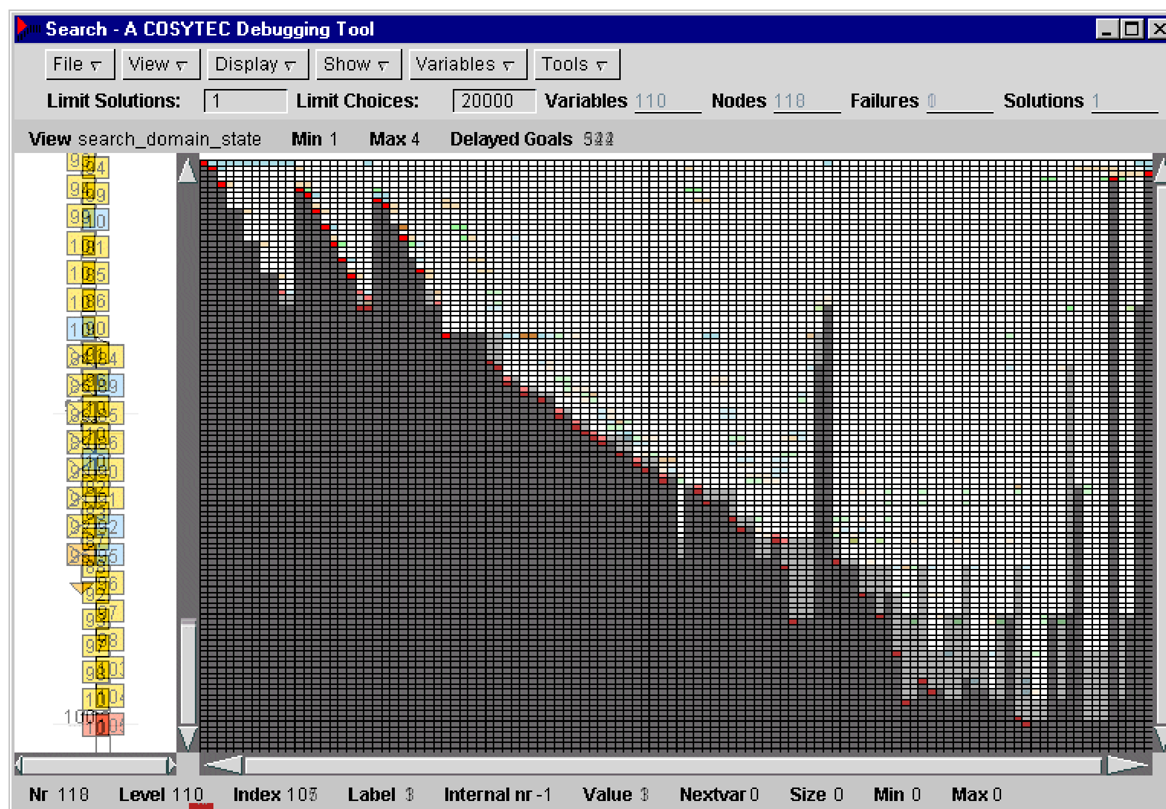


Figure 29: Overlay of two solutions

This type of overlay is quite useful to understand the differences between two similar programs, which are not always immediately visible in the two diagrams.

Just to finish the topic, we can also look at the incidence matrix of this extended problem. Do the redundant constraints also show a structure like the ones of the binary constraints?

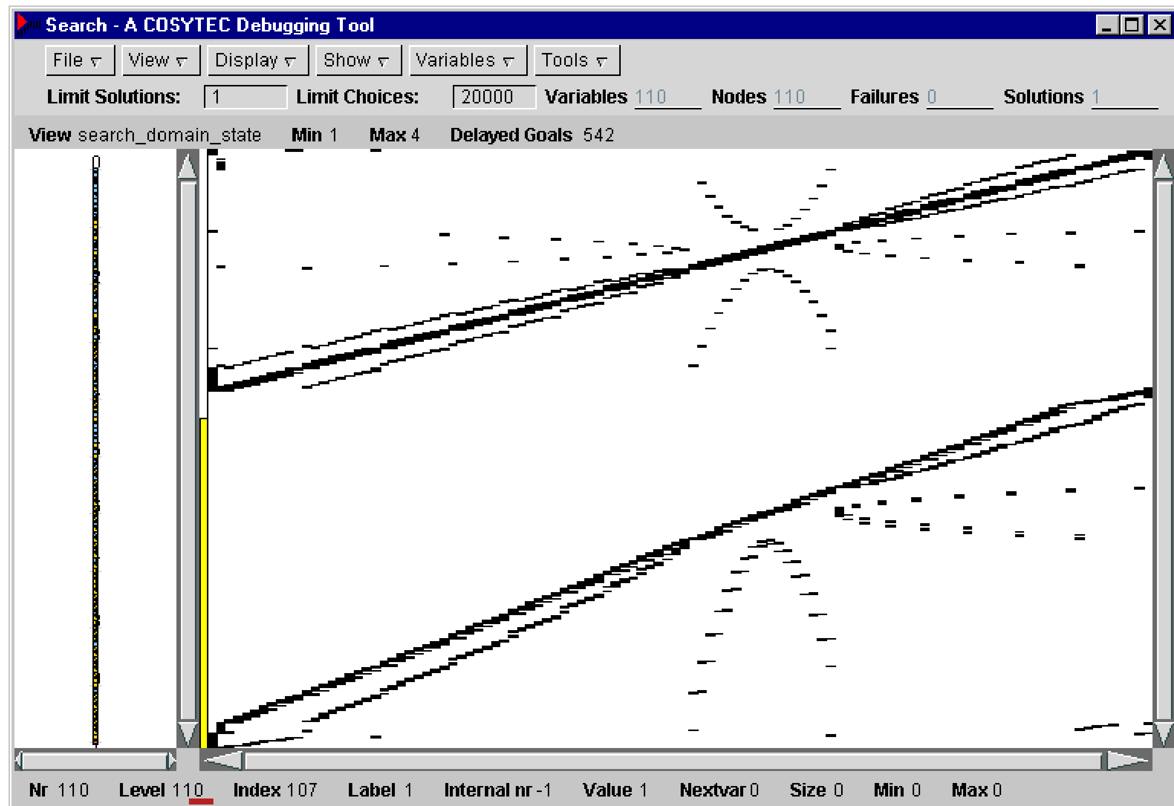


Figure 30: Incidence matrix with redundant constraints

They do indeed.

3.15 Summary

3.15.1 Simple map coloring problem

- limited constraint reasoning
- forward checking in disequality
- possible improvements by finding cliques

3.15.2 Small changes have big impact

- tie break in most constrained very important
- do not perturb static order of variables
- better a good static ordering than a bad dynamic one

3.15.3 Visualization helps to understand

- search tree shows thrashing
- deep backtracking after wrong choice

- domain update view shows variable selection
- incidence matrix useful to recognize structure
- overlay of diagrams to compare solutions

4 Ship loading

4.1 Problem

This demo shows how to model and solve a small scheduling problem. This example is taken from

- ROSEAUX, <u>Exercices et Problèmes Résolus de Recherche Opérationnelle - Tome 3</u>, Paris, 1983, p. 279-282.

The problem consists of scheduling the unloading and loading of a ship in a harbor. The different compartments must be first unloaded and the filled again with new cargo. For each operation, the duration and the resource required to perform this task is given. The figure below shows the list of all tasks.

Task nr	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
Duration	3	4	4	6	5	2	3	4	3	2	3	2	1	5	2	3	2
Resource Use	4	4	3	4	5	5	4	3	4	8	4	5	4	3	3	3	6

Task nr	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34
Duration	2	1	1	1	2	4	5	2	1	1	2	1	3	2	1	2	2
Resource Use	7	4	4	4	4	7	8	8	3	3	6	8	3	3	3	3	3

In order to relate these tasks to the compartments in the ship, we can look at the following diagram. An arrow point up-wards indicates a unloading operation, an arrow pointing downwards a loading operation. If there are several operations in one compartment, then they have to be performed in the right sequence.

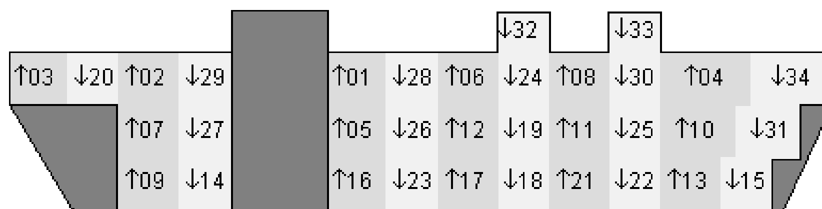


Figure 31: Ship loading example

Another constraint concerns the balancing of the ship. We can not for example completely empty the stern of the ship, while keeping the bow fully loaded. Taking these loading and unloading constraints into account, we come up with the following precedence graph.

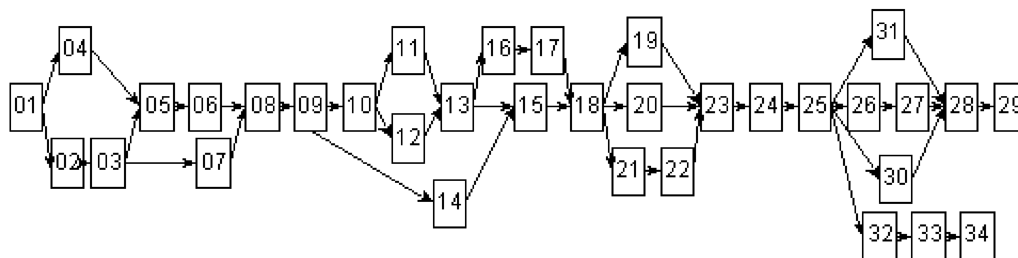


Figure 32: Precedence graph

This graph shows which tasks must be finished before another task can start. Some tasks can be run in parallel.

4.1.1 The Objective

The objective of the schedule is to finish all tasks as quickly as possible respecting a given resource limit. For different amounts of resource, different solutions will be obtained. We are also interested in finding the right balance between resource availability and project duration.

4.2 Model

Constraint programs are always expressed in terms of decision variables, the constraints over the decision variables which define possible solutions and a search method that is used to find values for the variables.

4.2.1 The variables

For each task, we have to decide the start time. These are the decision variables of our problem. As their initial domain we use a large time period, so that we are sure to find a solution respecting the precedence constraints. The duration and the resource requirement are fixed values given above in a table.

4.2.2 The precedence constraint

We can easily express the precedence conditions between tasks by inequality constraints. If some task B must run after another task A, we state the inequality constraint

```
start(b) = start(a)+duration(a).
```

4.2.3 The resource constraint

The resource constraint is expressed with a cumulative constraint. This constraint expresses for a set of tasks that at each time point the cumulated resource use does not exceed a given resource availability. Another parameter in the constraint gives us access to the overall end of the schedule. We use this value as our optimization criteria.

4.2.4 The search method

As the given problem is an optimization procedure, we use a minimization routine. To assign values to the variables we use a standard, built-in routine which selects the most-restricted variable first.

4.3 Program

To solve the problem in CHIP V5 Prolog we express the data for the problem in the predicate data/3. This is used to create a list of domain variables Sis, and domain variables Limit and End to be used by the cumulative constraint. The sequence of the tasks is represented in the data as a list of constraints and we use a feature of Prolog, meta-programming, to set up these constraints. In order to have direct access to the Si variables and Di variables we use the univ predicate to transform elements of the list into terms. The data for the inequality constraint is of the form After #= Before which means: task at position After should succeed the task at position Before. After stating the inequality we call cumulative to handle the manpower

conflict and min_max with the most_constrained heuristic to search for solutions to the problem.

```
top:-
  run(8, 100).

run(Upper, Last):-
  data(N, Dis, Mis),
  length(Sis, N),
  Sis :: 0..Last,
  Limit :: 0..Upper,
  End :: 0..Last,
  precedences(L),
  set_precedences(L, Sis, Dis),
  cumulative(Sis, Dis, Mis, unused, unused, Limit, End, unused),
  min_max(labeling(Sis), End),
  writeln(Sis).

labeling(Sis):-
  labeling(Sis, 0, most_constrained, indomain).

set_precedences(L, Sis, Dis):-
  Array_starts=..[starts|Sis], % starts(S1, S2, S3, ..)
  Array_durations=..[durations|Dis], % durations(D1, D2, D3, ..)
  set_pre_lp(L, Array_starts, Array_durations).

set_pre_lp([],_,_).
set_pre_lp([After #= Before|R], Array_starts, Array_durations):-
  arg(After, Array_starts, S2),
  arg(Before, Array_starts, S1),
  arg(Before, Array_durations, D1),
  S2 #= S1 + D1,
  set_pre_lp(R, Array_starts, Array_durations).

% nr of tasks, duration of tasks, resource use of tasks
data(34,[3,4,4,6,5,2,3,4,3,2,3,2,1,5,2,3,2,2,1,1,1,2,4,5,2,1,1,2,1,3,2,1,2,
2],
  [4,4,3,4,5,5,4,3,4,8,4,5,4,3,3,3,6,7,4,4,4,4,7,8,8,3,3,6,8,3,3,3,3,3]).

% after #= before, task indices
precedences([2#=1, 3#=2, 4#=1, 5#=3, 5#=4, 6#=5, 7#=3, 8#=6, 8#=7, 9#=8,
  10#=9, 11#=10, 12#=10, 13#=11, 13#=12, 14#=9, 15#=13, 15#=14, 16#=13,
  17#=16, 18#=15, 18#=17, 19#=18, 20#=18, 21#=18, 22#=21, 23#=19, 23#=20,
  23#=22, 24#=23, 25#=24, 26#=25, 27#=26, 28#=27, 28#=30, 28#=31, 29#=28,
  30#=25, 31#=25, 32#=25, 33#=32, 34#=33]).
```

4.4 Solution

The figure below shows the optimal solution with a resource limit of 8.

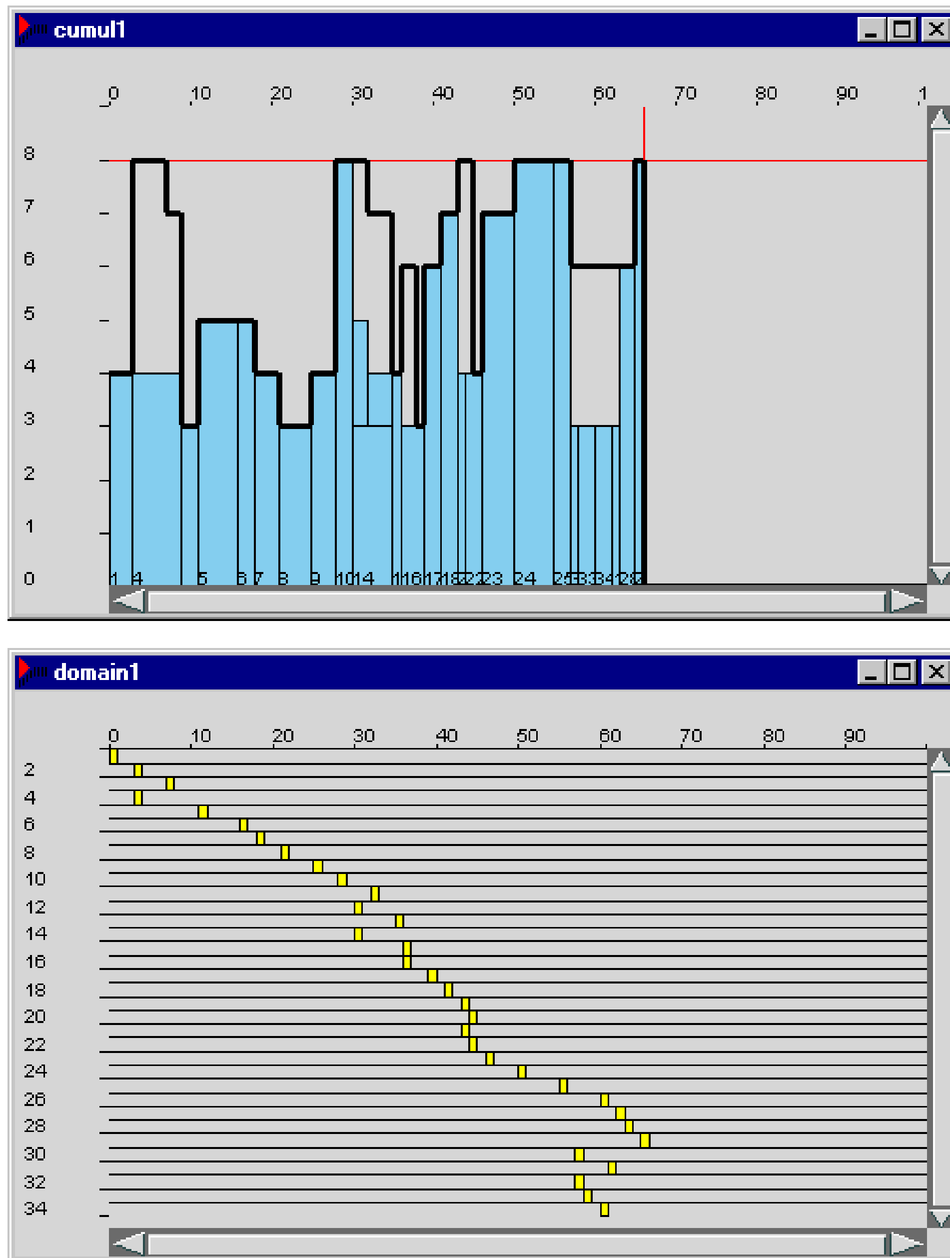


Figure 33: Visualization of solution

4.5 Searchtree

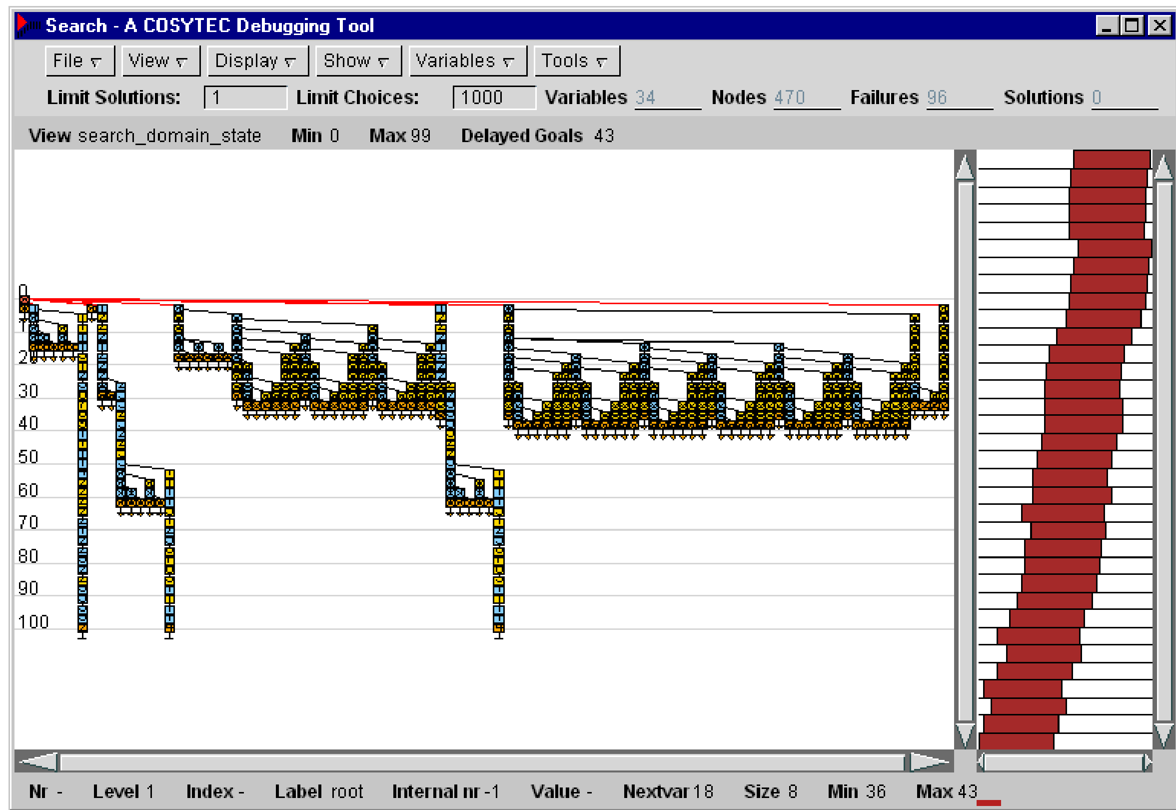


Figure 34: Searchtree

The picture shows the search tree generated by the program. We can identify three solutions being found within the min_max optimization. After each solution, the search restarts at the top. The proof of optimality is in the right part of the tree. We an also observe some thrashing behavior in two parts of the tree. Before finding the third solution, we explore a set of variables several times. The same happens in the proof of optimality.

4.6 Isomorphic subtrees

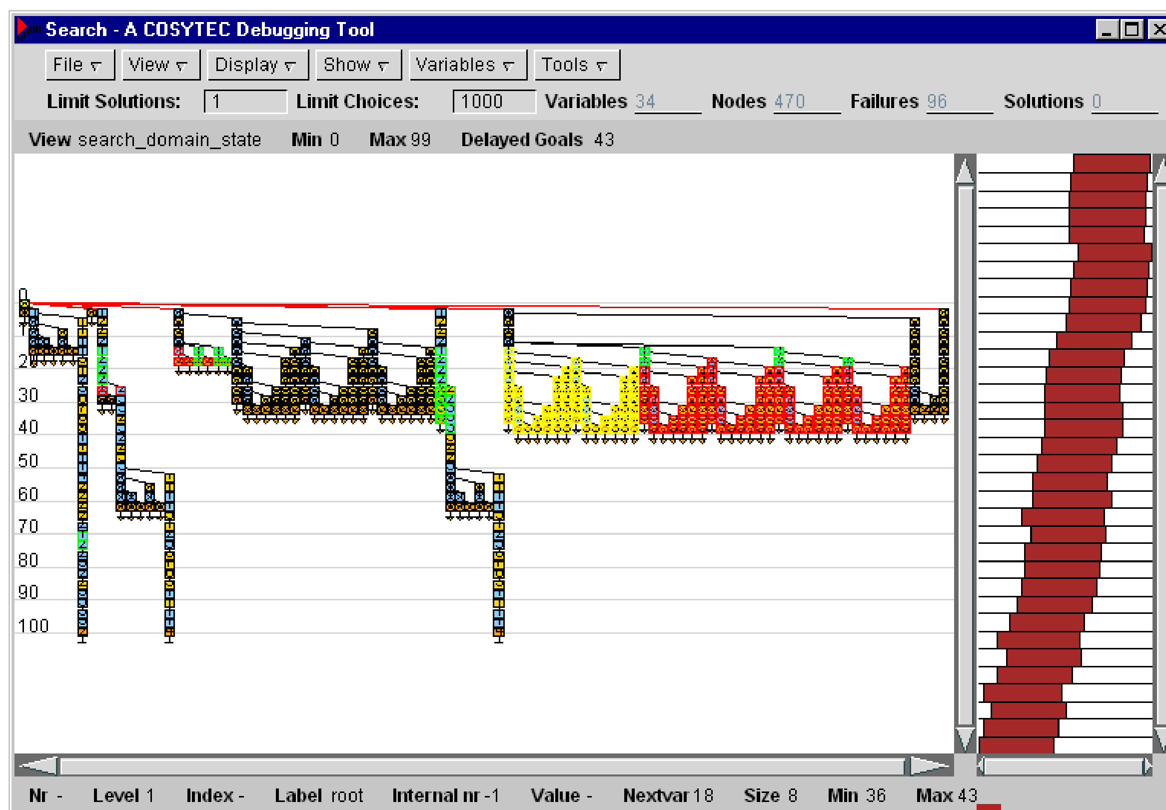


Figure 35: Isomorphic subtrees

We can use the tool to identify isomorphic sub-trees in the search tree tool to see that thrashing behavior. The yellow tree shows the part of the tree that is evaluated. We find two copies of that sub-tree, shown in red, and a number of other nodes which also assign these variables, but with different values. These nodes are bordered in green. Isomorphic sub-trees are an indication of thrashing. At some point in the tree we have made a wrong decision, so that we can not find any solution below. Due to a lack of propagation we do not detect this failure immediately. We repeatedly assign some other variables to the same values exploring all possible alternatives for that set before backtracking to the choice which initiated the problem.

4.7 Value Choice

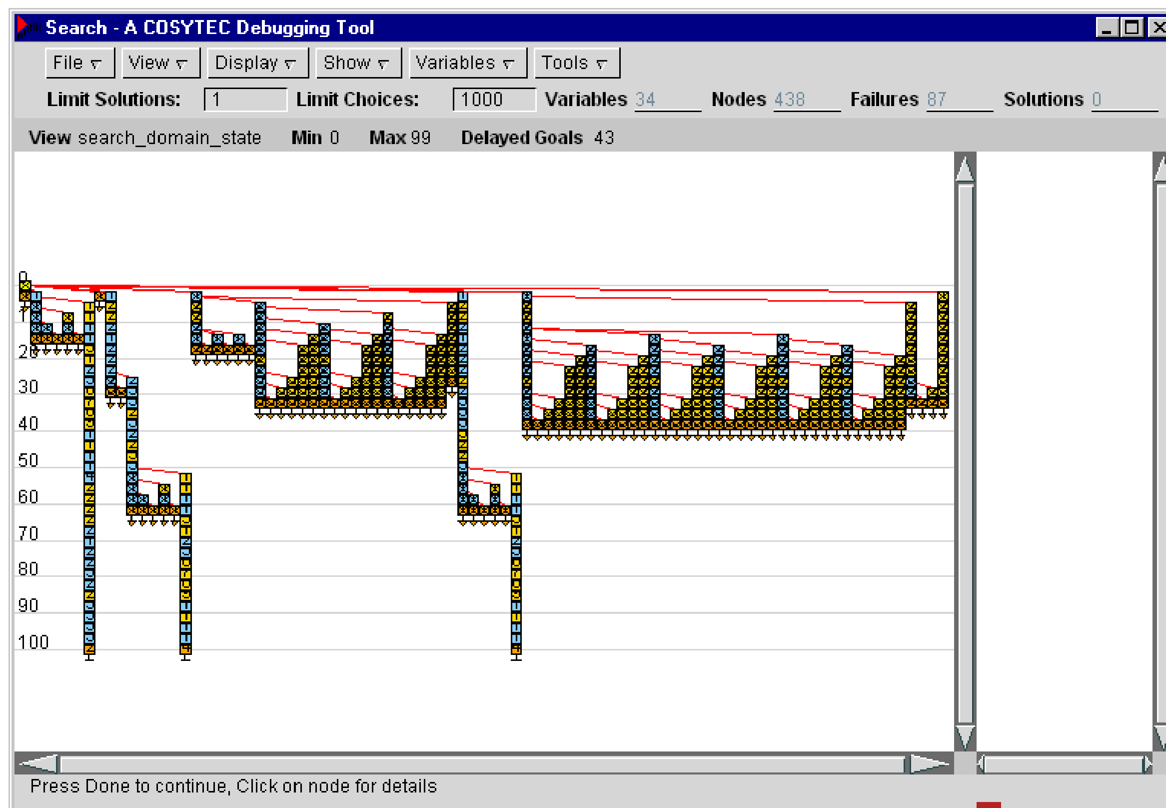


Figure 36: Value choice variant

A first change in the program would affect the assignment routine. The usual indomain/1 predicate tries to bind a variable to all values in its domain, starting from the smallest value. If a particular value is not possible, it will choose the next value in the domain, until the domain is exhausted. The important thing to note is that when a value fails in the assignment, it is not removed from the domain before the next value is tested. This can lead to some inefficiency, in particular in scheduling problems.

Often, the start dates of tasks are directly or indirectly linked to the cost of the schedule. If we do not remove the failed values from the domain, we will update the cost only when a value has been set. We might be able to detect a failure more rapidly, if we remove the inconsistent value before continuing with the enumeration, if the cost is updated by this removal. The predicate indomain1/1 actually performs this removal. If we replace indomain with indomain1 in the program, we obtain the tree shown above, which is quite similar, but slightly smaller (438 instead of 470 nodes).

4.8 Optimization

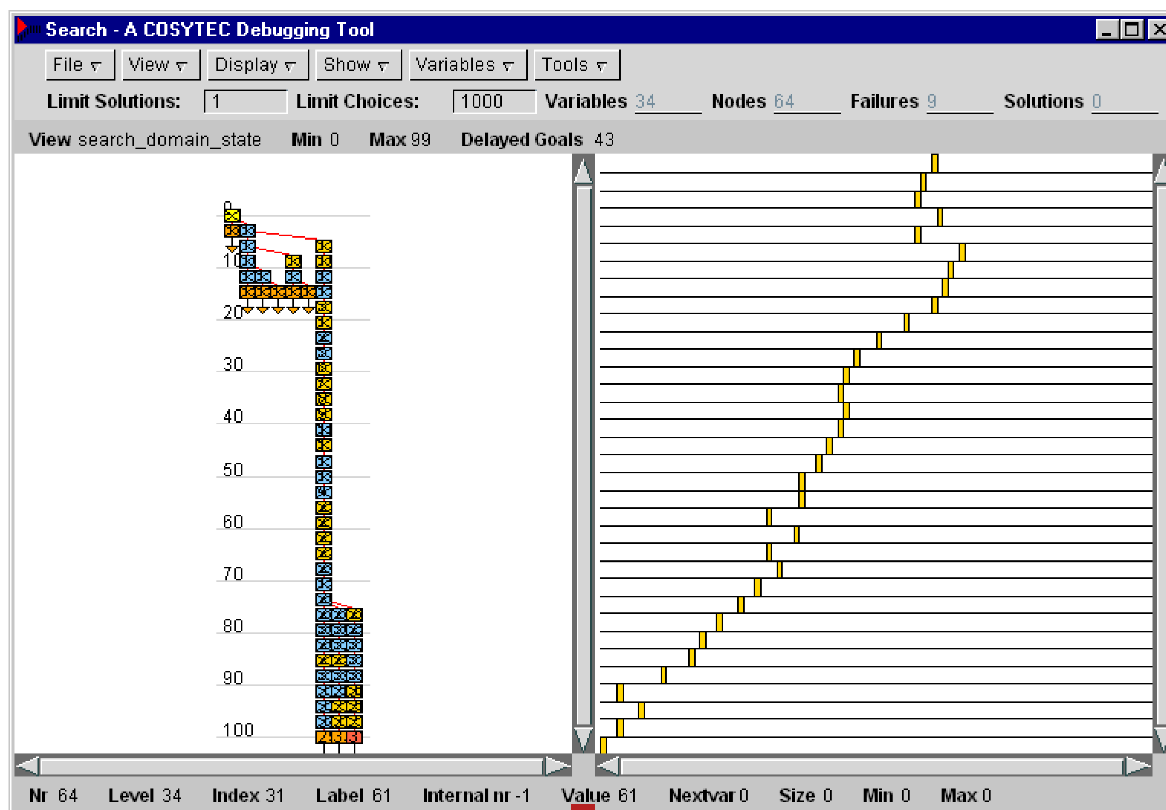


Figure 37: Solution with minimize

The next modification affects the optimization routine. We have used min_max as the optimization method. This meta-routine initially calls the generator predicate in its first argument in order to find a solution. Whenever a solution is found, it is stored with the new cost value. The program then backtracks to the start of the search, imposes a constraint on the cost (which should be below the last solution) and then restarts the search. If no solution is found, the last solution found is returned, if it exists. Otherwise, the program fails. The well-known disadvantage of this method is that the same part of the search space may be explored more than once. This can happen in particular if the cost estimation is not very good.

In CHIP, we also have an alternative optimization routine. The minimize constraint also tries to find a first solution. If it is found, the system continues the exploration of the search space and dynamically adds a constraint that the cost should be below the current best bound. In this way the search space is explored only once.

If we test the minimize routine on this example, we get a pleasant surprise: We now only need 64 nodes to find the optimum and prove optimality (instead of 438). If we compare the two trees, we can see that the tree is the same up to the point the first solution is found. This is not surprising, as the optimization routine does not affect the search for the first solution. But after the first solution, the trees differ dramatically. Is this just an example of minimize being more effective? No, there is something else. In the min_max, we restart the search from the beginning. But we do not follow the same path through the search space. As the upper bound has changed, many domains have been modified. Our dynamic variable selection chooses a different variable rather early in the tree. From then on, the search is quite

different. In the minimize search, we do not use the information about the cost to re-consider the variable selection. Only if we take a completely new branch in the tree, we will use this information. The min_max is more intelligent, but it does not work very well here.

4.9 Phaseline

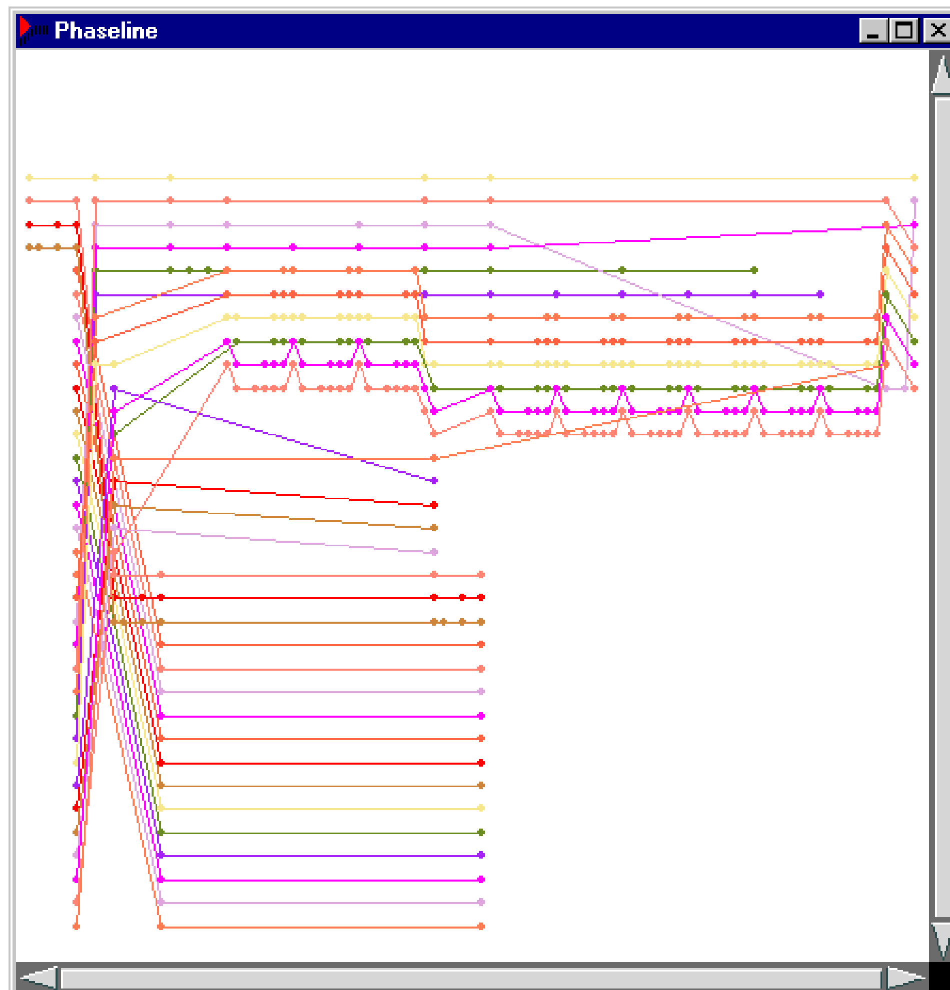


Figure 38: Phaseline display

How did we know that min_max is selecting the variables in a different sequence? There are two ways of finding this out. One uses the phase-line display which interprets the search tree in a novel way. Instead of connecting parent and children, we connect nodes which assign the same variables. We obtain a display like the one shown above.

Lines connect nodes in the search tree where the same variable is assigned. The colors are used to distinguish the different variables, they are repeated every 10 levels. We can see that after finding the first solution we completely reorganize the search. The variables which were assigned first in the initial solution are now assigned last, and two other variable set are enumerated first.

4.10 Cumulative View

The same information is available from the global constraint visualizer for the cumulative constraint. The first assignment step fixes a task in the middle of the time horizon.

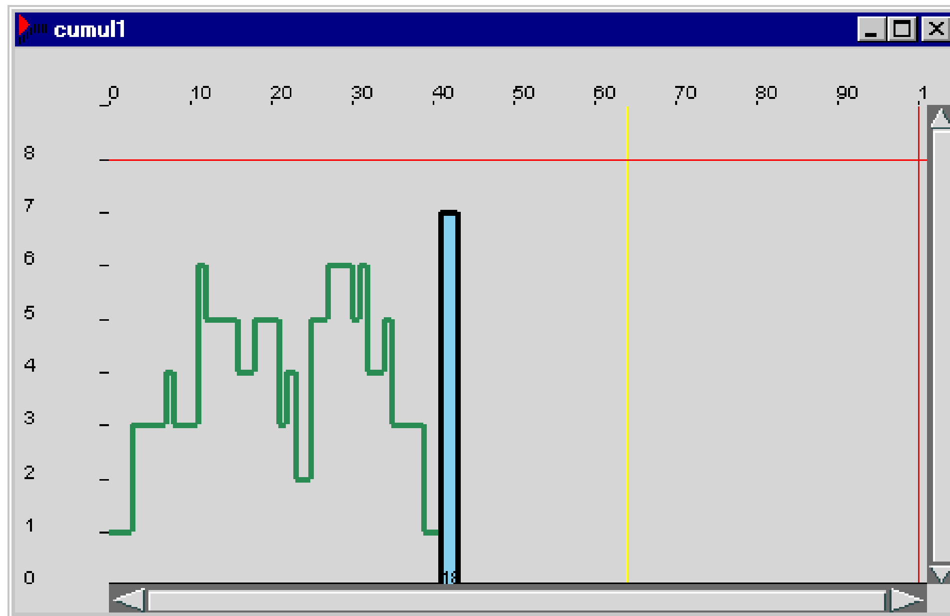


Figure 39: Cumulative profile after first step

For finding the first solution, the strategy selects next a task in the left part.

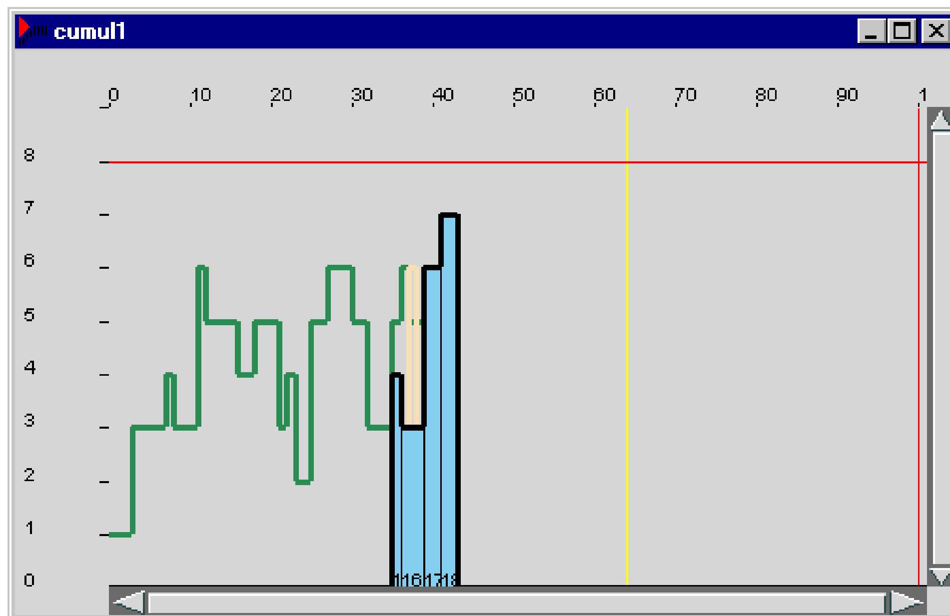


Figure 40: Cumulative profile after second step, minimize

The search continues to assign tasks in that part of the planning period, and only later switches to tasks to the right of the initial task.

When restarting after the first solution, the strategy selects a tasks to the right of the first task,

as this task is now more constrained than the one in the left part of the schedule.

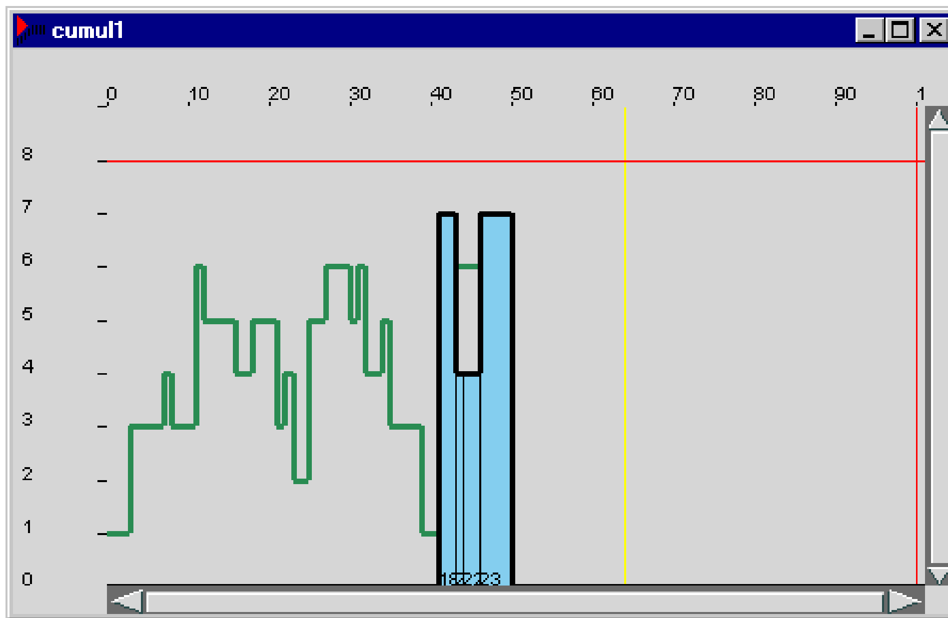


Figure 41: Cumulative profile after second step, min_max

This decision is not successful in this case and forces the system to spend more time in exploring this right part of the schedule in proving optimality.

4.11 Redundant Constraint

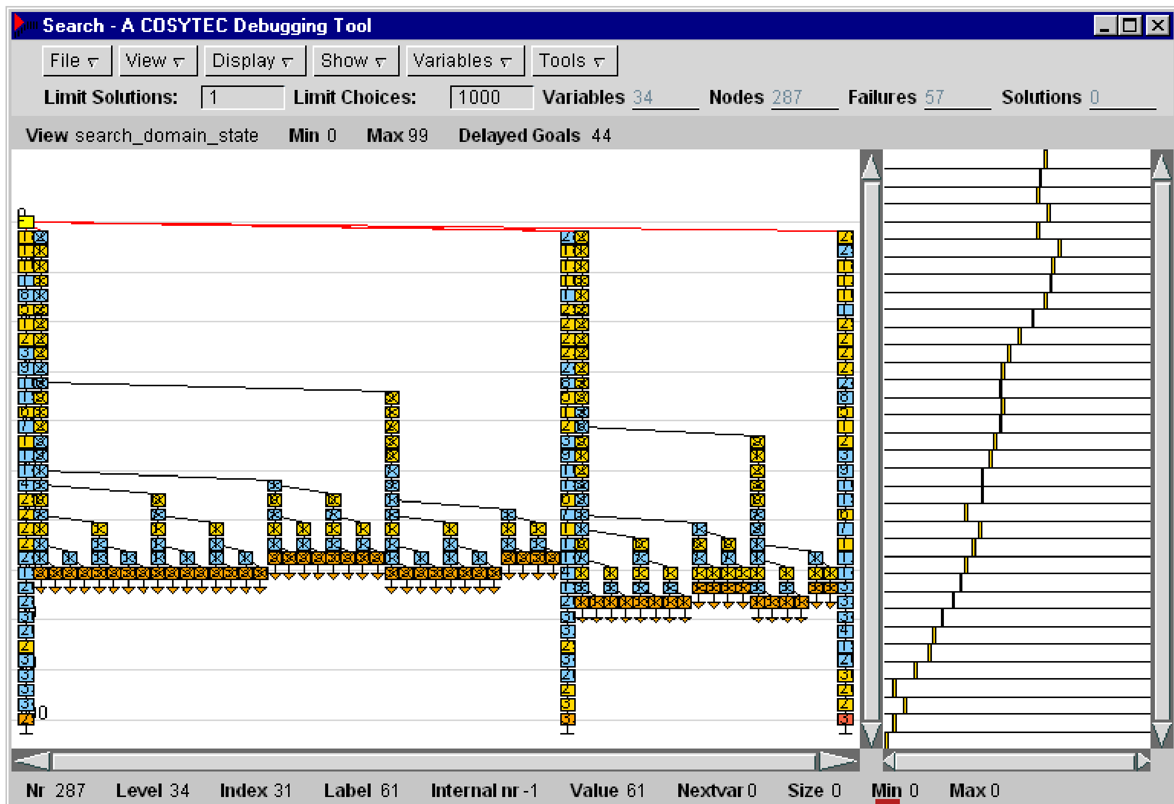


Figure 42: Redundant constraint

Another way of improving the program is to strengthen the constraint propagation. In the original model, we were using inequality constraints and one cumulative constraint. We can add to this a redundant precedence global constraint. In fact, the constraint is intended for this type of problem: It takes a set of precedence relations between tasks together with a set of cumulative resource constraints and calculates additional conditions from this combination of temporal and resource constraints. In order to use the constraint in our program, we add the lines:

```
        ...
prec_uses(Use,Uses),
        prec_prec(L,Precs),
        precedence([Upper],Start,Dur,Uses,Precs),
        ...
```

in the main routine and also add the predicates:

```
prec_uses([],[]).
prec_uses([H|T],[[H]|R]):-
        prec_uses(T,R).

prec_prec([],[]).
prec_prec([B #= A|R],[[A,B]|S]):-
        prec_prec(R,S).
```

The improved reasoning should lead to a smaller search tree. Looking at the solution with the min_max optimization and indomain, this is the case. We now need 287 nodes instead of 470. But the solution looks quite different.

4.12 Initial Domains

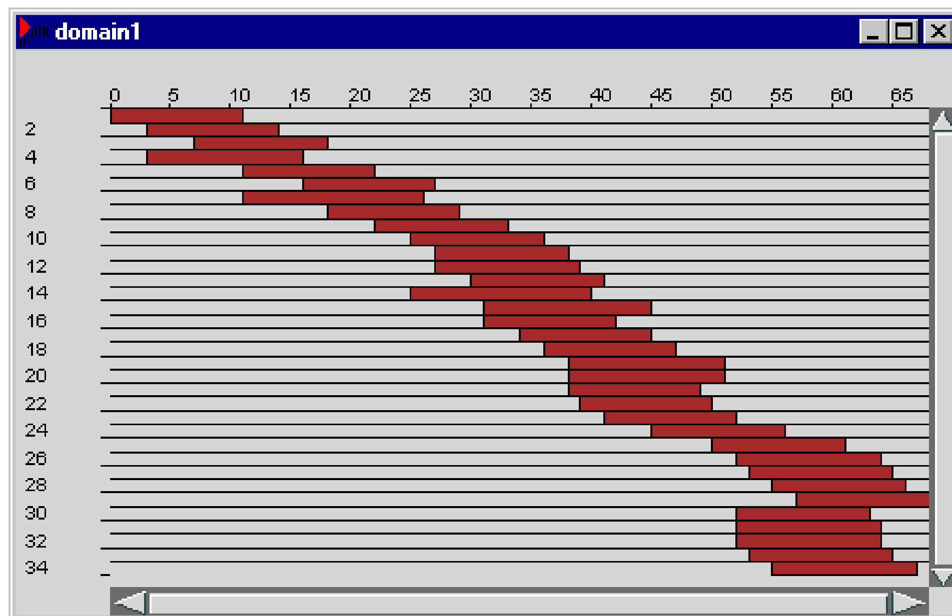


Figure 43: Initial domain without precedence

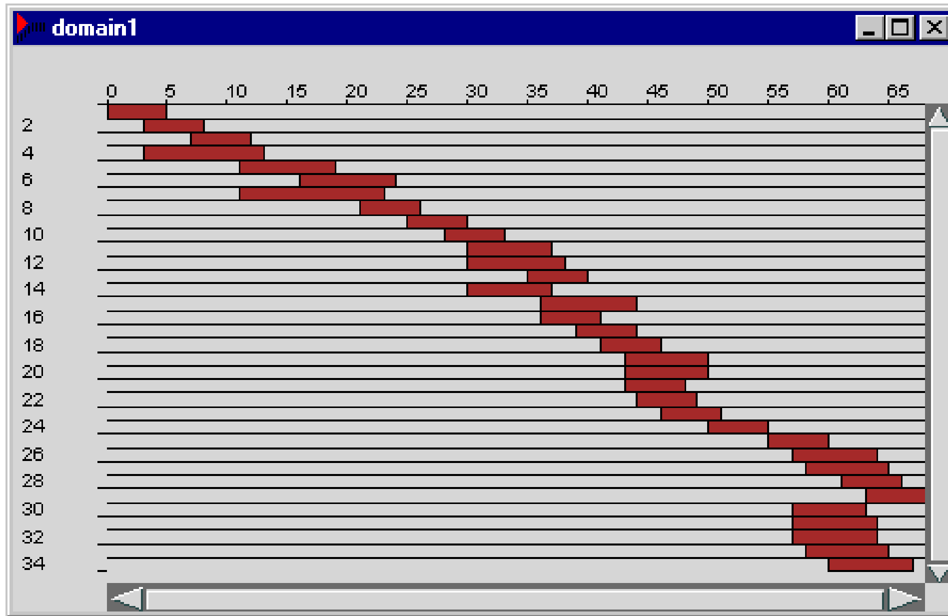


Figure 44: Initial domain with precedence

4.13 Combination

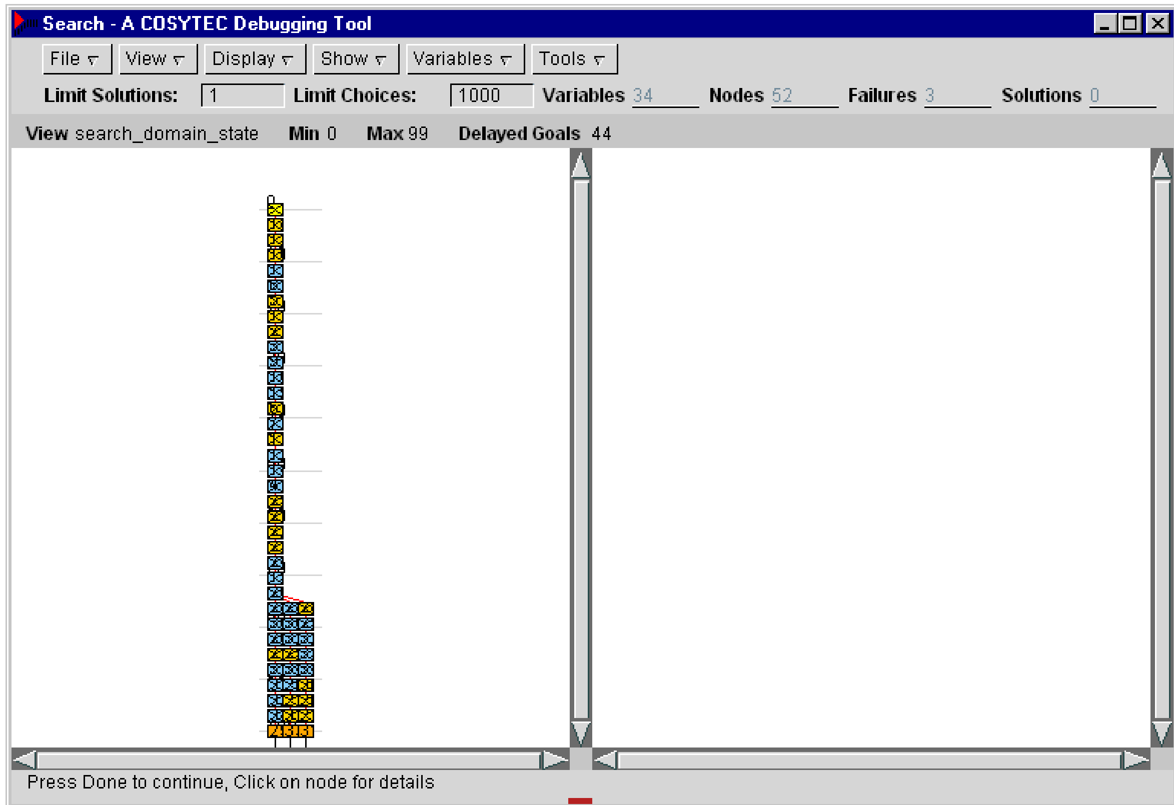


Figure 45: Combination of techniques

4.14 Summary

4.14.1 Small Scheduling Problem

- one cumulative resource constraint
- precedence redundant constraint
- set of inequality constraints

4.14.2 Optimization variants

- very different form of search tree
- difference not due to meta program
- impact on variable ordering more important

4.14.3 Improvements

- indomain1 as alternative choice
- different optimization meta-strategy
- redundant constraint to improve propagation

5 Cutting Stock

5.1 Problem

A detailed description of the problem can be found in [DSV88]. We just recall the main principles. The problem arises in the furniture manufacturing where rectilinear pieces of wood must be cut into smaller pieces. The demand of each type of piece is given, and we try to satisfy the demand with minimal cost. Figure 1 shows the size of the pieces and their demand.

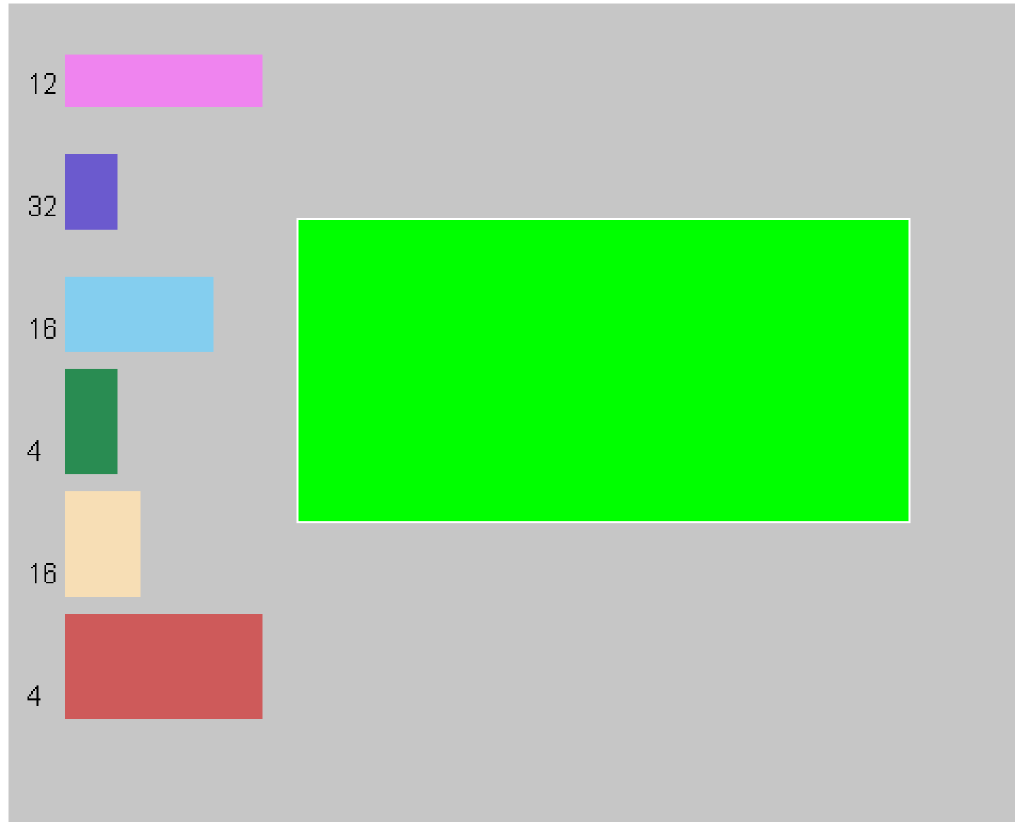


Figure 46: Cutting problem

Due to the technique of cutting, the number of possible cut combinations is severely limited. The overall approach to solving the problem is to generate all possible cut combinations and then to select a combination of patterns that satisfies the demand. We will here not consider the problem of generating the cut pattern, but figure 2 shows all possible 72 patterns that we have to consider.

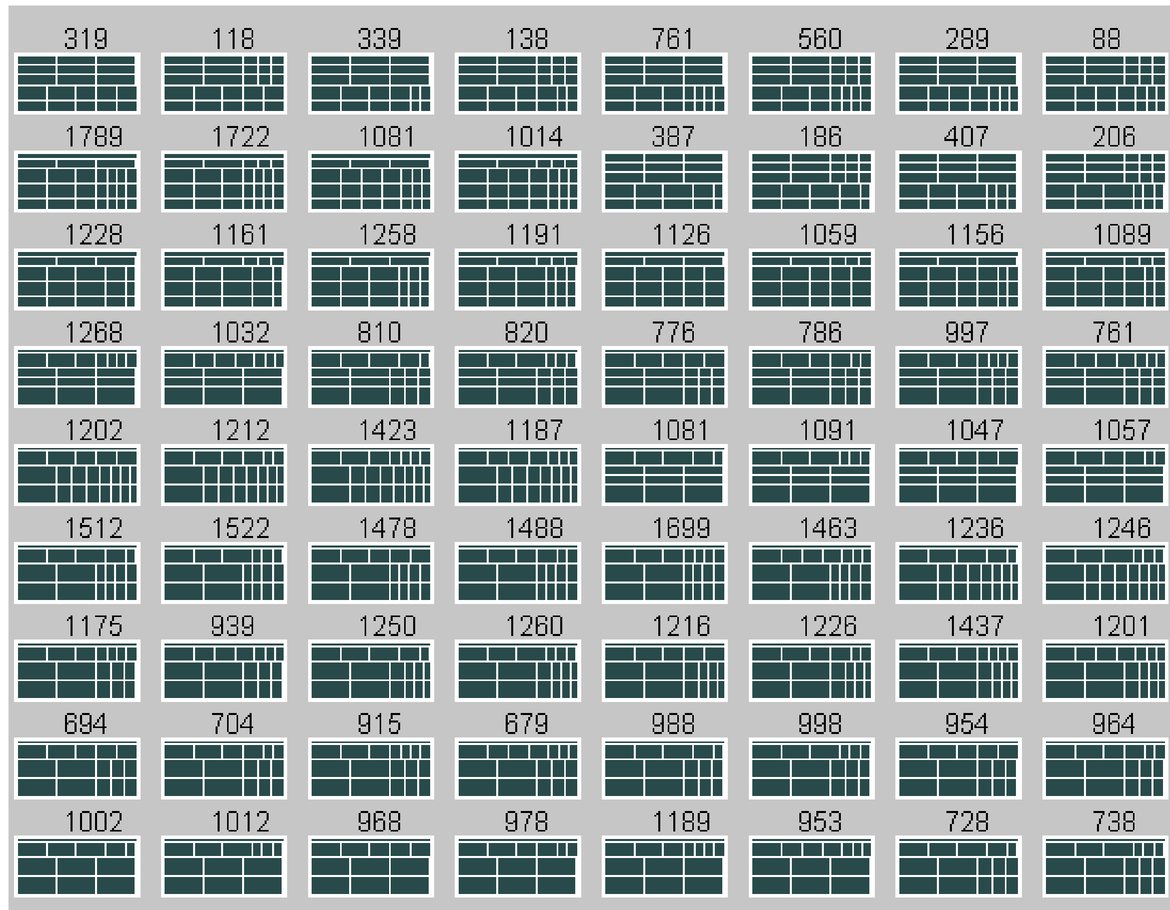


Figure 47: All possible combinations

For each pattern, we now the amount of waste, that is material that is not used for generated pieces. We also know the number of items of different types in each pattern. Given the demand, it is clear that four pattern should be chosen to satisfy the demand. If for some item type, we produce more than the required number, we can use these pieces for stock, and they are not counted as waste. The colored items in figure 2 give an example solution, with the read pattern selected twice and the green pattern selected once each. In order to obtain a simpler constraint model, we may prohibit to use one pattern more than once, but this may change the optimal solution.

5.2 Model

This model is quite straightforward and the corresponding CHIP program is given below. As we have 72 possible pattern, we introduce 72 variables which state if we use the pattern or not. The value 0 denotes that we do not use the pattern, the value 1 states that we use the pattern. As we know that we will use exactly four pattern, we state the sum of the variables is equal to 4. With similar constraints, we state that the demand for each item type must be satisfied. The coefficients of the variables are the number of items produced for each pattern. We can also calculate the total amount of waste by adding up the waste of each pattern that is selected. This means that we can express the complete problem just using linear arithmetic constraints.

5.3 Program

```
top(Y,Cost):-
        model_01(Y,Cost),
        min_max(labeling(Y,0,input_order,indomain),Cost).

model_01(Y,Cost):-
Y = [Y1,Y2,Y3,Y4,Y5,Y6,Y7,Y8,Y9,
Y10,Y11,Y12,Y13,Y14,Y15,Y16,Y17,Y18,Y19,
Y20,Y21,Y22,Y23,Y24,Y25,Y26,Y27,Y28,Y29,
Y30,Y31,Y32,Y33,Y34,Y35,Y36,Y37,Y38,Y39,
Y40,Y41,Y42,Y43,Y44,Y45,Y46,Y47,Y48,Y49,
Y50,Y51,Y52,Y53,Y54,Y55,Y56,Y57,Y58,Y59,
Y60,Y61,Y62,Y63,Y64,Y65,Y66,Y67,Y68,Y69,
Y70,Y71,Y72],

Y :: 0..1,
Cost :: 0:100000,

Y1 + Y2 + Y3 + Y4 + Y5 + Y6 + Y7 + Y8 + Y9 + Y10
 + Y11 + Y12 + Y13 + Y14 + Y15 + Y16 + Y17 + Y18 + Y19 + Y20
 + Y21 + Y22 + Y23 + Y24 + Y25 + Y26 + Y27 + Y28 + Y29 + Y30
 + Y31 + Y32 + Y33 + Y34 + Y35 + Y36 + Y37 + Y38 + Y39 + Y40
 + Y41 + Y42 + Y43 + Y44 + Y45 + Y46 + Y47 + Y48 + Y49 + Y50
 + Y51 + Y52 + Y53 + Y54 + Y55 + Y56 + Y57 + Y58 + Y59 + Y60
 + Y61 + Y62 + Y63 + Y64 + Y65 + Y66 + Y67 + Y68 + Y69 + Y70
 + Y71 + Y72 #= 4,
Cost #= 1002 * Y1 + 1012 * Y2 + 968 * Y3 + 978 * Y4 + 1189 * Y5
 + 953 * Y6 + 728 * Y7 + 738 * Y8 + 694 * Y9 + 704 * Y10 + 915 * Y11
 + 679 * Y12 + 988 * Y13 + 998 * Y14 + 954 * Y15 + 964 * Y16 + 1175 *
Y17 + 939 * Y18 + 1250 * Y19 + 1260 * Y20 + 1216 * Y21 + 1226 * Y22
+ 1437 * Y23 + 1201 * Y24 + 1512 * Y25 + 1522 * Y26 + 1478 * Y27 + 1488 *
Y28 + 1699 * Y29 + 1463 * Y30 + 1236 * Y31 + 1246 * Y32 + 1202 *
Y33 + 1212 * Y34 + 1423 * Y35 + 1187 * Y36 + 1081 * Y37 + 1091 * Y38
+ 1047 * Y39 + 1057 * Y40 + 1268 * Y41 + 1032 * Y42 + 810 * Y43 + 820
* Y44 + 776 * Y45 + 786 * Y46 + 997 * Y47 + 761 * Y48 + 1228 * Y49 +
 1161 * Y50 + 1258 * Y51 + 1191 * Y52 + 1126 * Y53 + 1059 * Y54 + 1156
 * Y55 + 1089 * Y56 + 1789 * Y57 + 1722 * Y58 + 1081 * Y59 + 1014 * Y60 +
387 * Y61 + 186 * Y62 + 407 * Y63 + 206 * Y64 + 319 * Y65 + 118
* Y66 + 339 * Y67 + 138 * Y68 + 761 * Y69 + 560 * Y70 + 289 * Y71 +
88 * Y72,
6 * Y1 + 6 * Y2 + 6 * Y3 + 6 * Y4 + 6 * Y5 + 6
 * Y6 + 4 * Y7 + 4 * Y8 + 4 * Y9 + 4 * Y10 + 4 * Y11 + 4 * Y12 + 4
 * Y13 + 4 * Y14 + 4 * Y15 + 4 * Y16 + 4 * Y17 + 4 * Y18 + 4 * Y19
 + 4 * Y20 + 4 * Y21 + 4 * Y22 + 4 * Y23 + 4 * Y24 + 4 * Y25 + 4 *
Y26 + 4 * Y27 + 4 * Y28 + 4 * Y29 + 4 * Y30 + 2 * Y31 + 2 * Y32 + 2
 * Y33 + 2 * Y34 + 2 * Y35 + 2 * Y36 + 3 * Y37 + 3 * Y38 + 3 * Y39
 + 3 * Y40 + 3 * Y41 + 3 * Y42 + 2 * Y43 + 2 * Y44 + 2 * Y45 + 2 *
Y46 + 2 * Y47 + 2 * Y48 #= 4,
1 * Y1 + 3 * Y3 + 2 * Y4 + 1 * Y5 +
3 * Y6 + 7 * Y7 + 6 * Y8 + 9 * Y9 + 8 * Y10 + 7 * Y11 + 9 * Y12 +
5 * Y13 + 4 * Y14 + 7 * Y15 + 6 * Y16 + 5 * Y17 + 7 * Y18 + 3 * Y19
 + 2 * Y20 + 5 * Y21 + 4 * Y22 + 3 * Y23 + 5 * Y24 + 1 * Y25 + 3 *
Y27 + 2 * Y28 + 1 * Y29 + 3 * Y30 + 7 * Y31 + 6 * Y32 + 9 * Y33 +
8 * Y34 + 7 * Y35 + 9 * Y36 + 1 * Y37 + 3 * Y39 + 2 * Y40 + 1 * Y41
 + 3 * Y42 + 4 * Y43 + 3 * Y44 + 6 * Y45 + 5 * Y46 + 4 * Y47 + 6 *
Y48 + 3 * Y49 + 3 * Y50 + 9 * Y53 + 9 * Y54 + 6 * Y55 + 6 * Y56 +
3 * Y57 + 3 * Y58 + 9 * Y59 + 9 * Y60 + 2 * Y61 + 2 * Y62 + 6 * Y65
 + 6 * Y66 + 4 * Y67 + 4 * Y68 + 2 * Y69 + 2 * Y70 + 6 * Y71 + 6 *
Y72 #= 16,
2 * Y13 + 2 * Y14 + 2 * Y15 + 2 * Y16 + 2 * Y17 + 2 * Y18 + 4 * Y19 + 4 *
```

```
Y20 + 4 * Y21 + 4 * Y22 + 4 * Y23 + 4 * Y24 + 6
* Y25 + 6 * Y26 + 6 * Y27 + 6 * Y28 + 6 * Y29 + 6 * Y30 + 6 * Y31
+ 6 * Y32 + 6 * Y33 + 6 * Y34 + 6 * Y35 + 6 * Y36 #= 4,
3 * Y1 + 3
* Y2 + 2 * Y3 + 2 * Y4 + 2 * Y5 + 1 * Y6 + 3 * Y7 + 3 * Y8 + 2 *
Y9 + 2 * Y10 + 2 * Y11 + 1 * Y12 + 3 * Y13 + 3 * Y14 + 2 * Y15 + 2
* Y16 + 2 * Y17 + 1 * Y18 + 3 * Y19 + 3 * Y20 + 2 * Y21 + 2 * Y22
+ 2 * Y23 + 1 * Y24 + 3 * Y25 + 3 * Y26 + 2 * Y27 + 2 * Y28 + 2 * Y29 + 1 *
Y30 + 3 * Y31 + 3 * Y32 + 2 * Y33 + 2 * Y34 + 2 * Y35 + 1
* Y36 + 3 * Y37 + 3 * Y38 + 2 * Y39 + 2 * Y40 + 2 * Y41 + 1 * Y42
+ 3 * Y43 + 3 * Y44 + 2 * Y45 + 2 * Y46 + 2 * Y47 + 1 * Y48 + 9 * Y49 + 9 *
Y50 + 9 * Y51 + 9 * Y52 + 6 * Y53 + 6 * Y54 + 6 * Y55 + 6
* Y56 + 6 * Y57 + 6 * Y58 + 3 * Y59 + 3 * Y60 + 6 * Y61 + 6 * Y62
+ 6 * Y63 + 6 * Y64 + 4 * Y65 + 4 * Y66 + 4 * Y67 + 4 * Y68 + 4 * Y69 + 4 *
Y70 + 2 * Y71 + 2 * Y72 #= 16,
1 * Y1 + 3 * Y2 + 2 * Y4 +
 3 * Y5 + 3 * Y6 + 1 * Y7 + 3 * Y8 + 2 * Y10 + 3 * Y11 + 3 * Y12 +
 1 * Y13 + 3 * Y14 + 2 * Y16 + 3 * Y17 + 3 * Y18 + 1 * Y19 + 3 * Y20 + 2 *
Y22 + 3 * Y23 + 3 * Y24 + 1 * Y25 + 3 * Y26 + 2 * Y28 + 3 *
 Y29 + 3 * Y30 + 1 * Y31 + 3 * Y32 + 2 * Y34 + 3 * Y35 + 3 * Y36 +
 1 * Y37 + 3 * Y38 + 2 * Y40 + 3 * Y41 + 3 * Y42 + 7 * Y43 + 9 * Y44 + 6 *
Y45 + 8 * Y46 + 9 * Y47 + 9 * Y48 + 3 * Y49 + 6 * Y50 + 9 *
 Y51 + 12 * Y52 + 3 * Y54 + 6 * Y55 + 9 * Y56 + 9 * Y57 + 12 * Y58
 + 9 * Y59 + 12 * Y60 + 2 * Y61 + 11 * Y62 + 6 * Y63 + 15 * Y64 + 9
* Y66 + 4 * Y67 + 13 * Y68 + 6 * Y69 + 15 * Y70 + 6 * Y71 + 15 * Y72 #= 32,
6 * Y37 + 6 * Y38 + 6 * Y39 + 6 * Y40 + 6 * Y41 + 6 * Y42
 + 4 * Y43 + 4 * Y44 + 4 * Y45 + 4 * Y46 + 4 * Y47 + 4 * Y48 + 3 *
Y49 + 2 * Y50 + 3 * Y51 + 2 * Y52 + 3 * Y53 + 2 * Y54 + 3 * Y55 + 2
 * Y56 + 3 * Y57 + 2 * Y58 + 3 * Y59 + 2 * Y60 + 9 * Y61 + 6 * Y62
 + 9 * Y63 + 6 * Y64 + 9 * Y65 + 6 * Y66 + 9 * Y67 + 6 * Y68 + 9 *
Y69 + 6 * Y70 + 9 * Y71 + 6 * Y72 #= 12.
```

5.4 Solution

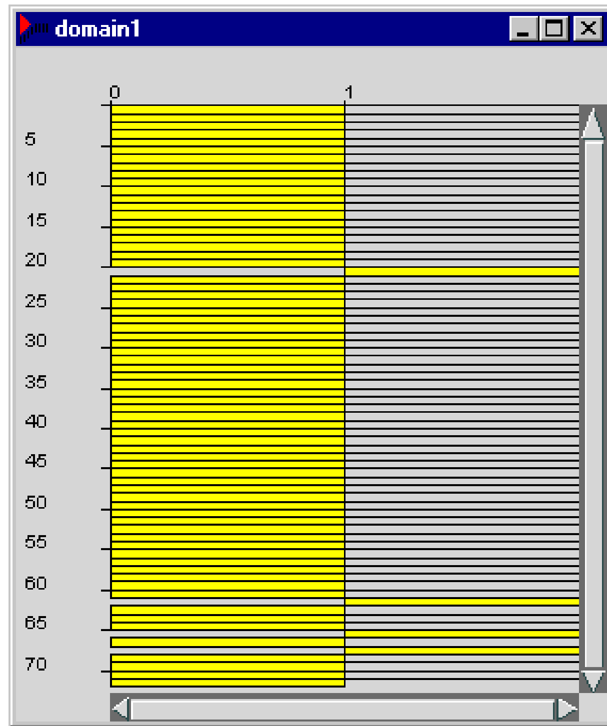


Figure 48: Solution

5.5 Searchtree

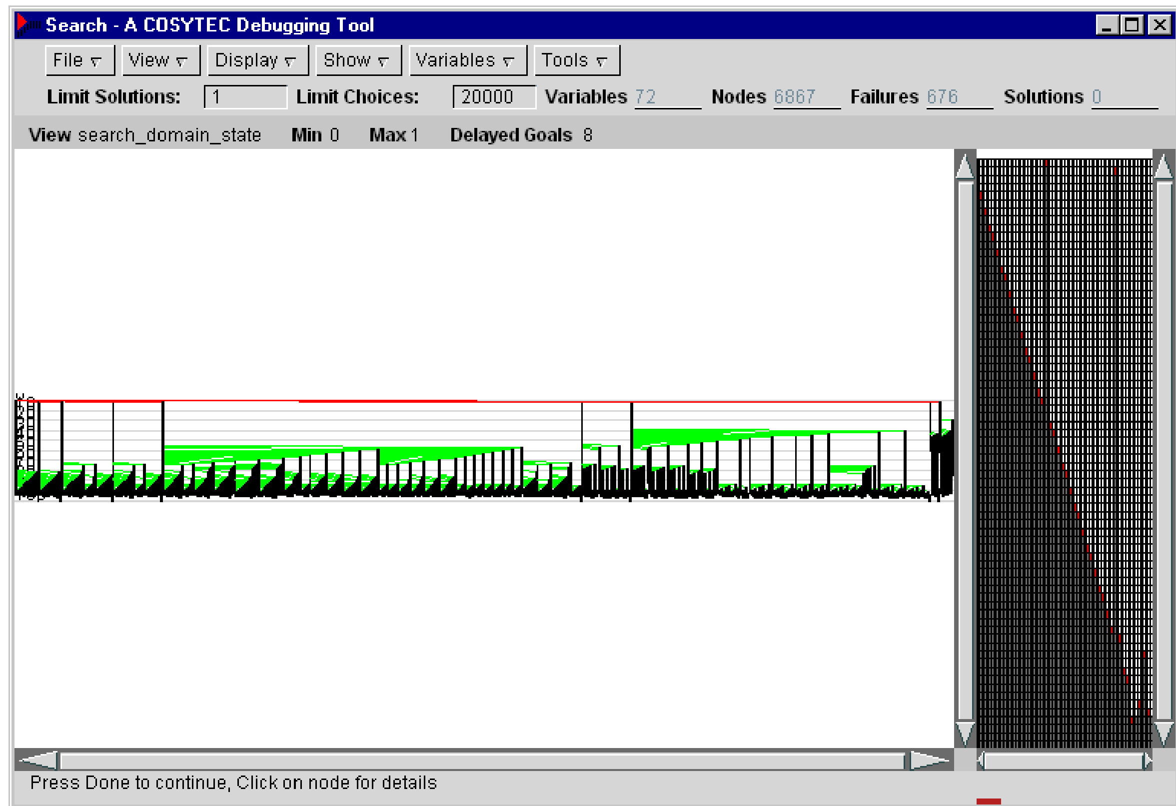


Figure 49: Initial searchtree

5.6 Analysis

5.6.1 Bad Initial Solution

The first thing we can notice in the search tree is the number of improving solutions that we find. The table below gives the cost of the different solutions and the number of the node in which they are found.

Node	Cost
224	3325
427	3315
838	3302
1229	3282
4170	2380
4611	2370

6384	1668
6564	1658

We can see that the initial solution is roughly twice the cost of the optimal one.

5.6.2 Backtracking before initial solution

Before finding the first solution (without a cost limit), the system already starts backtracking. The constraint propagation can not be very strong!

5.6.3 Deep Backtracking

As the optimization progresses, the backtracking occurs higher and higher in the tree.

5.6.4 Implication from Cost

In the proof of optimality, after the optimal solution has been found, three variables are forced to be one by constraint propagation. Only one variable is left to be instantiated. We use quite a lot of nodes to find that there is no variable which can improve the cost.

5.6.5 Rediscovery

Looking at isomorphic copies of the first solution, we notice that the initial assignment is repeated in all branches of the tree.

5.7 Optimization

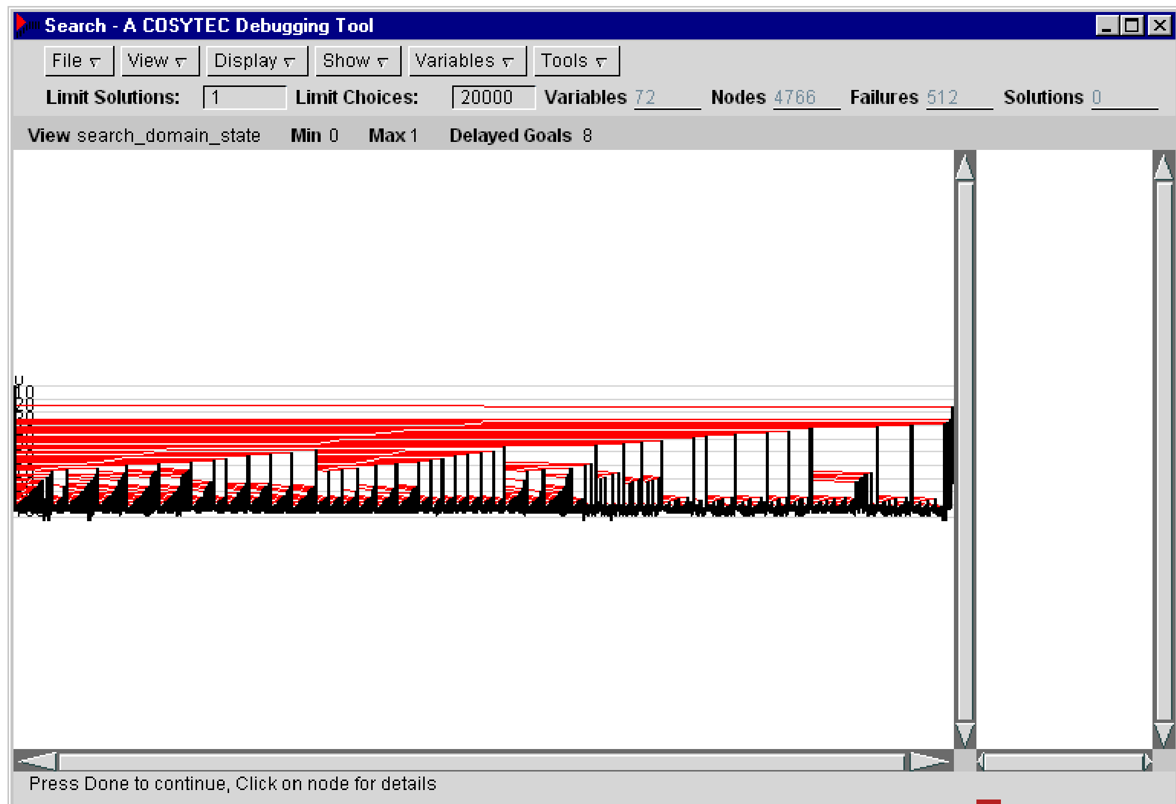


Figure 50: Searchtree with minimize

If we use the minimize predicate search for the optimal solution, we find a reduction of the number of search nodes from 6847 to 4766 nodes. This reduction by one third is good, but the resulting tree is still far to big.

5.8 Redundant Constraints

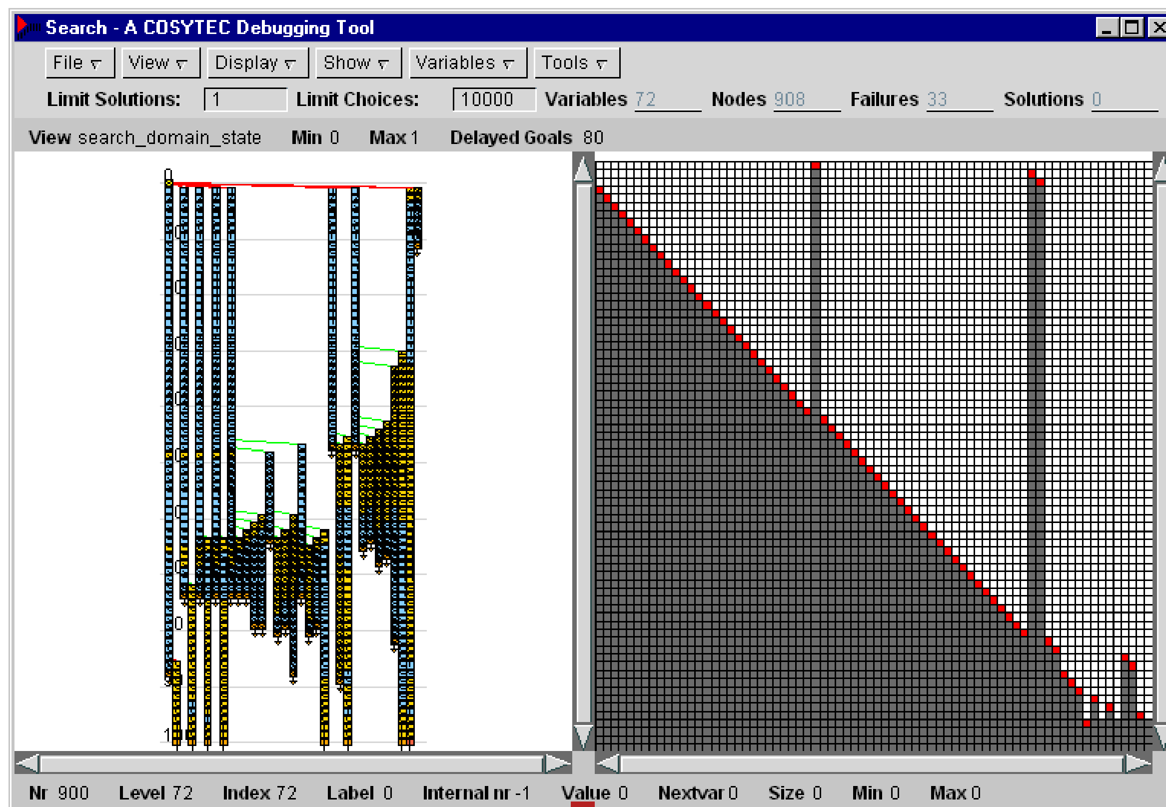


Figure 51: Searchtree with redundant constraint

In which way can we improve the 0/1 model? As the domains consist of only two values, we can not easily devise dynamic variable selection or assignment strategies. On the constraint side, we can combine all arithmetic constraint in a linear model. The Simplex algorithm (respectively the Gaussian elimination) keeps ensures consistency of the constraints with the partial assignment. Expressing the linear model in CHIP is quite simple. Instead of defining the variables as domain variables, we create them as linear variables within the range 0 to 1. We use the linear constraint equality symbols (^=, ^=) to express the constraints. But how do we link the finite domain variables and the variables of the linear relaxation?

The easiest way is a co-routine which links the two sets of decision variables. As soon as one of them is instantiated, it is bound to the other variable. As the search is performed in the finite domain model, more information is transferred from the finite domain model to the linear model, but the inverse information is important to reduce the search space. The program below shows the interaction.

```
top(X,Cost):-
        model_01(X,Cost,Terms),
```

```
        linear_model(Y,Lcost),
        mix(X,Y),
        connect(Cost,Lcost),
        min_max(labeling(X,0,input_order,indomain),Cost).

mix([],[]).
mix([X|X1],[Y|Y1]):-
        mix1(X,Y),
        mix1(Y,X),
        mix(X1,Y1).

?-delay mix1(ground,any).
mix1(X,X).
```

The link between the two cost variables is only performed in one direction. If a new bound on the cost variable of the finite domain model is stated, this bound is transferred to the linear model. In CHIP, this connection is done with a touched demon, shown in the program below.

```
connect(C,Cr):-
        touched(update,C,Cr,max).

update(C,Cr):-
        domain_info(C,Min,Max,_,_,_),
        Cr ^<= Max.
```

Note how the search tree has changed! We still find the same number of solutions, starting from a rather bad initial cost value. But at each step, the constraints perform a much better pruning, so that a lot less nodes must be explored. This is visible in particular for finding the first solution and the proof of optimality.

We can also compare the number of nodes which are required to find each solution. In the table below we compare the results from the original program and our modified version.

Node (old)	Node (new)	Cost
224	74	3325
427	148	3315
838	228	3302
1229	308	3282
4170	538	2380
4611	643	2370
6384	828	1668
6564	900	1658

5.9 Alternative

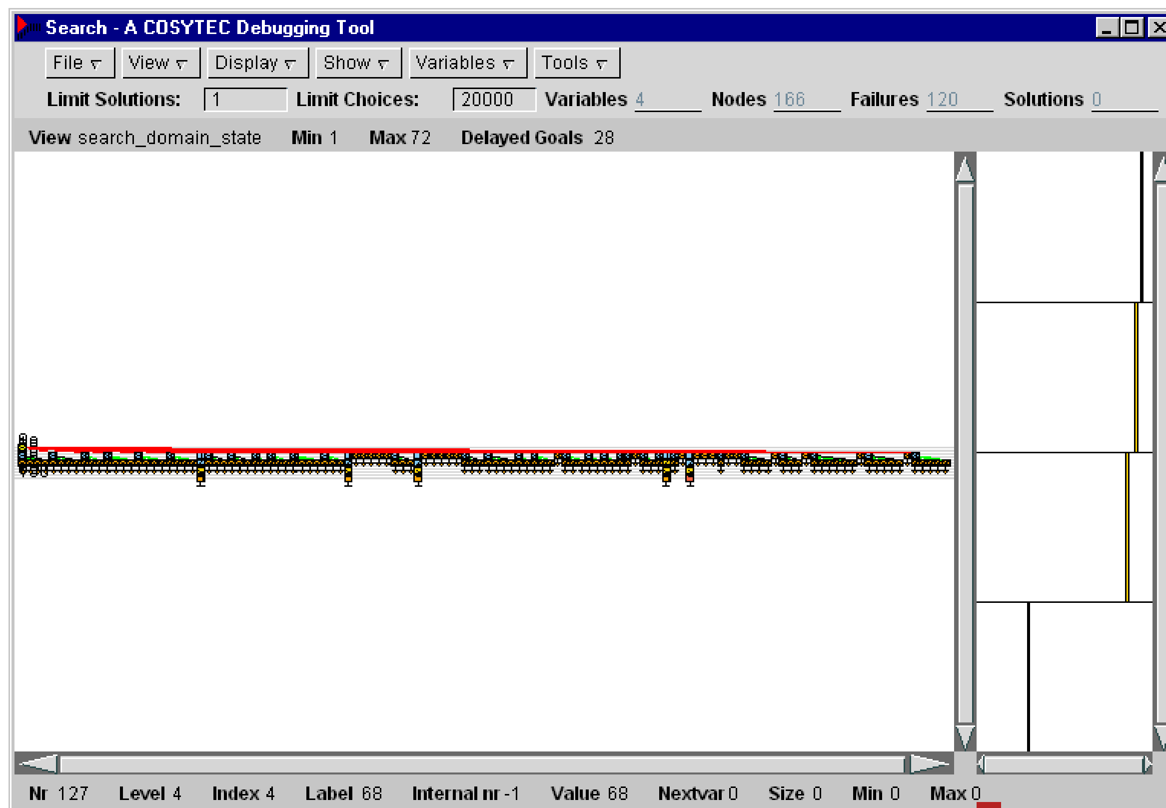


Figure 52: First finite domain model

The finite domain model looks at the problem in the inverse way. Instead of deciding whether we use a pattern or not, we decide which pattern to use. This means that we have four variables which range of a domain of 1 to 72. For each variable an element constraint expresses the waste depending on the selected pattern. The total waste is equal to the sum of these cost variables. In a similar way, the demand constraints are expressed with element constraints and inequality constraints.

5.10 Program

```
?-lib search.

top:-
        top(_,_).

top(X,Cost):-
        model(X,Cost,Terms),
        search_number(X,K),
        search_start(X,min_max(labeling(K,1,input_order,assign),Cost),[choi
ces<-20000,winw<-760,winh<-500,cost=Cost]).

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

model([X1,X2,X3,X4],Cost,[t(1,X1,C1),t(2,X2,C2),t(3,X3,C3),t(4,X4,C4)]) :-
```

```
        [X1,X2,X3,X4] :: 1..72,
        Cost :: 0:100000,
L=
[1002,1012,968,978,1189,953,728,738,694,704,915,679,988,998,954,
964,1175,939,1250,1260,1216,1226,1437,1201,1512,1522,1478,1488,
1699,1463,1236,1246,1202,1212,1423,1187,1081,1091,1047,1057,1268,
1032,810,820,776,786,997,761,1228,1161,1258,1191,1126,1059,1156,
1089,1789,1722,1081,1014,387,186,407,206,319,118,339,138,761,560,
289,88],

QL1                                                                    =
[6,6,6,6,6,6,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,2,2,2,2,2,2,3,
3,3,3,3,3,2,2,2,2,2,2,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],

QL2 =
[1,0,3,2,1,3,7,6,9,8,7,9,5,4,7,6,5,7,3,2,5,4,3,5,1,0,3,2,1,3,7,6,9,8,7,9,1,
0,3,2,1,3,4,3,6,5,4,6,3,3,0,0,9,9,6,6,3,3,9,9,2,2,0,0,6,6,4,4,2,2,6,6],

QL3 =
[0,0,0,0,0,0,0,0,0,0,0,0,2,2,2,2,2,2,4,4,4,4,4,4,6,6,6,6,6,6,6,6,6,6,6,6,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],

QL4 =
[3,3,2,2,2,1,3,3,2,2,2,1,3,3,2,2,2,1,3,3,2,2,2,1,3,3,2,2,2,1,3,3,2,2,2,1,3,
3,2,2,2,1,3,3,2,2,2,1,9,9,9,9,6,6,6,6,6,6,3,3,6,6,6,6,4,4,4,4,4,4,2,2],

QL5 =
[1,3,0,2,3,3,1,3,0,2,3,3,1,3,0,2,3,3,1,3,0,2,3,3,1,3,0,2,3,3,1,3,0,2,3,3,1,
3,0,2,3,3,7,9,6,8,9,9,3,6,9,12,0,3,6,9,9,12,9,12,2,11,6,15,0,9,4,13,6,15,6,
15],

QL6 =
[0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,6,
6,6,6,6,6,4,4,4,4,4,4,3,2,3,2,3,2,3,2,3,2,3,2,9,6,9,6,9,6,9,6,9,6,9,6],

element(X1,L,C1),
element(X2,L,C2),
element(X3,L,C3),
element(X4,L,C4),

Cost #= C1 + C2 + C3 + C4,

element(X1,QL1,Q11),
element(X2,QL1,Q12),
element(X3,QL1,Q13),
element(X4,QL1,Q14),
element(X1,QL2,Q21),
element(X2,QL2,Q22),
element(X3,QL2,Q23),
element(X4,QL2,Q24),
element(X1,QL3,Q31),
element(X2,QL3,Q32),
element(X3,QL3,Q33),
element(X4,QL3,Q34),
element(X1,QL4,Q41),
element(X2,QL4,Q42),
element(X3,QL4,Q43),
element(X4,QL4,Q44),
element(X1,QL5,Q51),
element(X2,QL5,Q52),
element(X3,QL5,Q53),
```

```
element(X4,QL5,Q54),
element(X1,QL6,Q61),
element(X2,QL6,Q62),
element(X3,QL6,Q63),
element(X4,QL6,Q64),

Q11 + Q12 + Q13 + Q14 #= 4,
Q21 + Q22 + Q23 + Q24 #= 16,
Q31 + Q32 + Q33 + Q34 #= 4,
Q41 + Q42 + Q43 + Q44 #= 16,
Q51 + Q52 + Q53 + Q54 #= 32,
Q61 + Q62 + Q63 + Q64 #= 12.

?-top.
```

5.11 Symmetry Removal

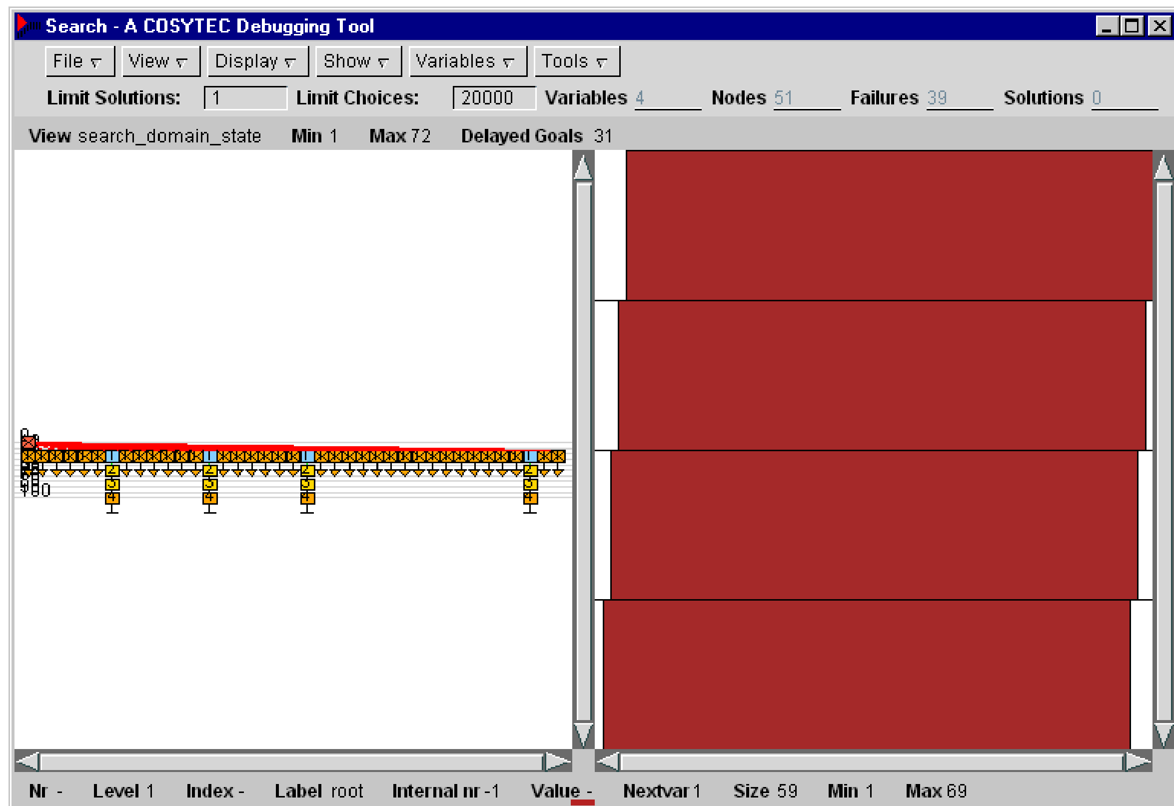


Figure 53: Searchtree with symmetry removal

The finite domain model also contains symmetrical solutions, as any permutation of a solution is again a solution with the same cost. If we don't remove this symmetry, we will find the same solution several times, basically wasting computational effort.

To remove the symmetry, we can enforce an order of the decision variables by adding inequality constraints.

```
X1 #< X2,
X2 #< X3,
X3 #< X4,
```

This symmetry removal is used in the demo program above. You can see the impact on the

initial domain, which is shown in the right window.

Instead of removing the symmetry on the decision variables, we may want to impose constraints on the cost variables. This will help to impose a lower bound on the total cost by stating the inequalities

```
C1 #<= C2,
C2 #<= C3,
C3 #<= C4,
```

Note that we have to use smaller or equal constraints in this case, as the costs of two different pattern may be the same. Since we do not use the strict inequality, we may still consider some symmetrical solutions. This overhead will be balanced against the good lower bound cost propagation that this form of symmetry removal offers.

5.12 Labeling on Cost

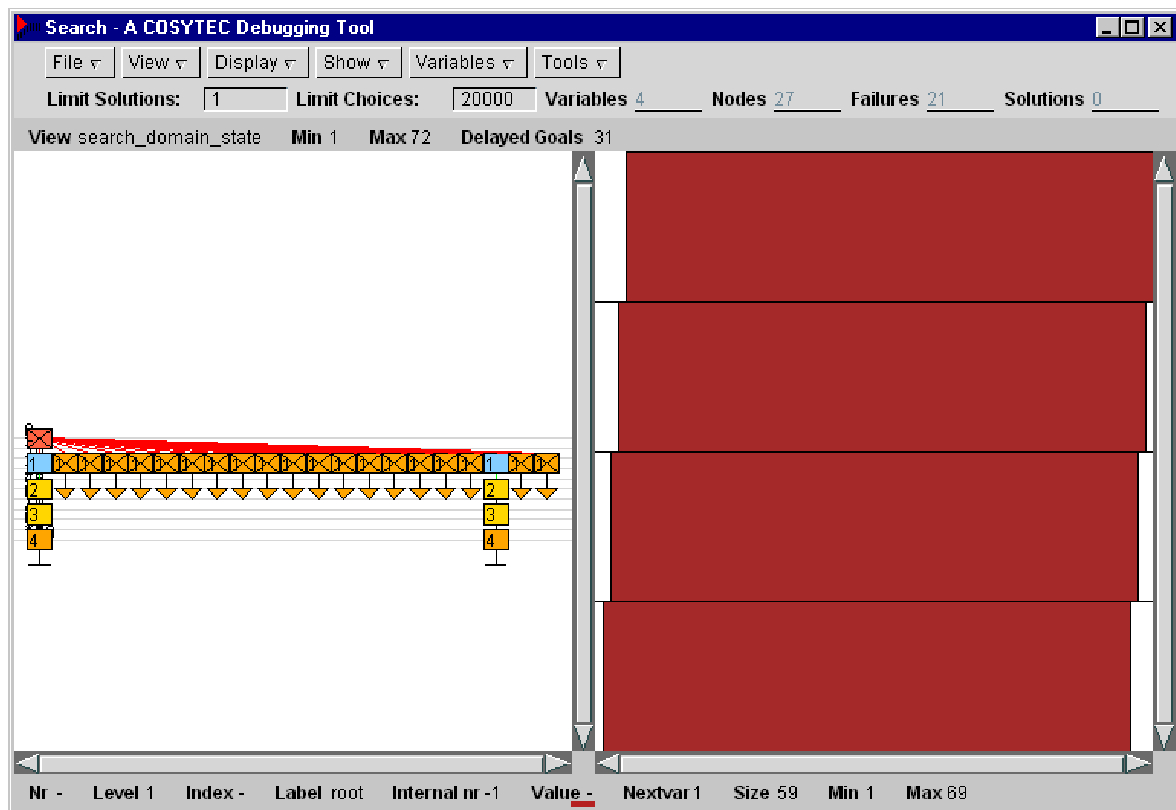


Figure 54: Labeling on cost

Another standard method in the search strategy consists in not labeling the decision variables themselves, but first to assign the cost variables and them the decision variables. This way, the decision variables will be selected in order of increasing cost, which is a good strategy to find solutions with small overall cost. Program below shows the CHIP program.

```
top(X,Cost):-
        model(X,Cost,Terms),
        min_max(labeling(Terms,2,input_order,assign),Cost).
```

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

5.13 Domain Splitting

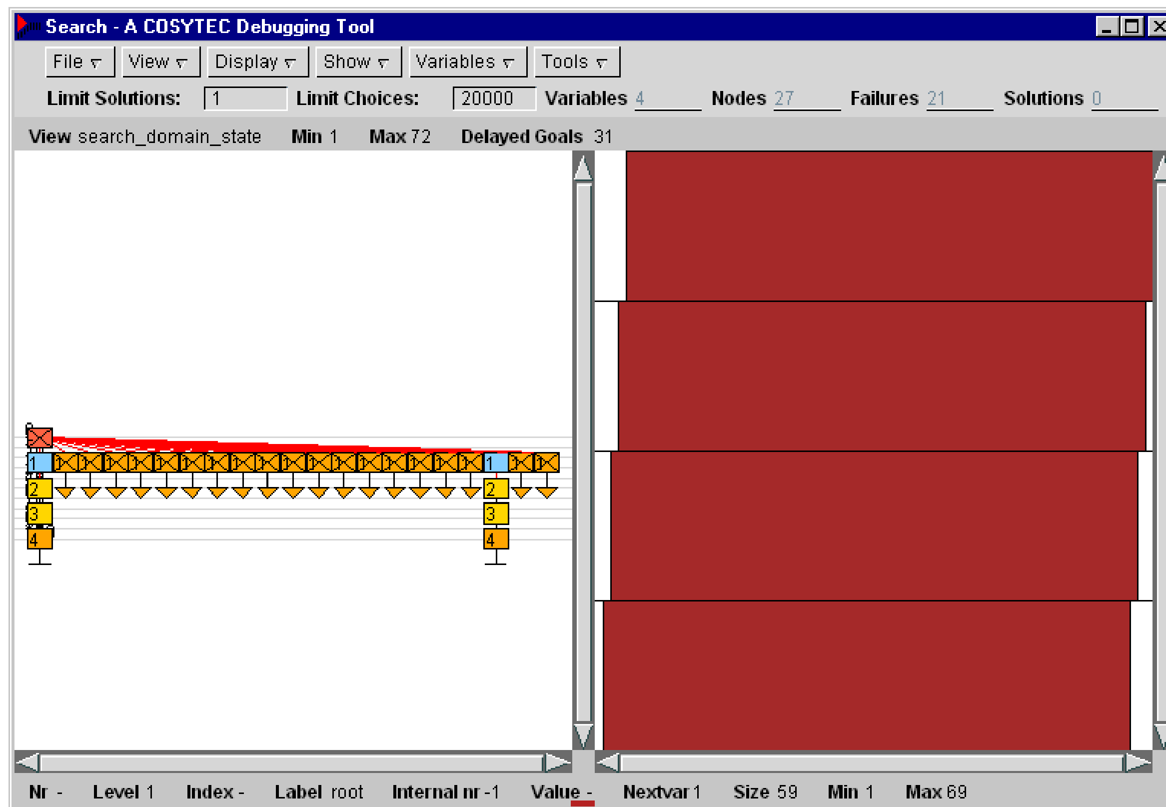


Figure 55: Domain splitting

To reduce the number of choices even further, we can apply a domain splitting strategy instead of an indomain assignment. At each step of the search we assign one variable. But instead of testing each value independently, we can split the domain into two equal parts by introducing an inequality constraint. After this branching, we further reduce the domain of that variable until it is instantiated. By splitting the domain, we hope to avoid enumeration of values which are not consistent with the constraint store. The program for the domain splitting is given below.

```
top(X,Cost):-
        model(X,Cost,Cost_variables),
        min_max(labeling(Cost_variables,0,input_order,split),Cost).

split(X):-
        integer(X),
        !.
split(X):-
        domain_info(X,Min,Max,_,_,_),
        Mid is (Min+Max)/2,
        split(X,Mid).

split(X,Mid):-
        X #<= Mid,
```

```
        split(X).
split(X,Mid):-
        X # Mid,
        split(X).
```

Note that we use the splitting operation on the cost variables, not on the decision variables. In this way the added inequality constraint will have an immediate effect on the total cost.

The domain splitting variant again dramatically reduces the execution time. The reason for this is that the system does not have to test individual values separately, but can remove a large number of alternatives in one test. This improvement is not visible in the search tree tool, as immediate failures are not shown.

5.14 Combination

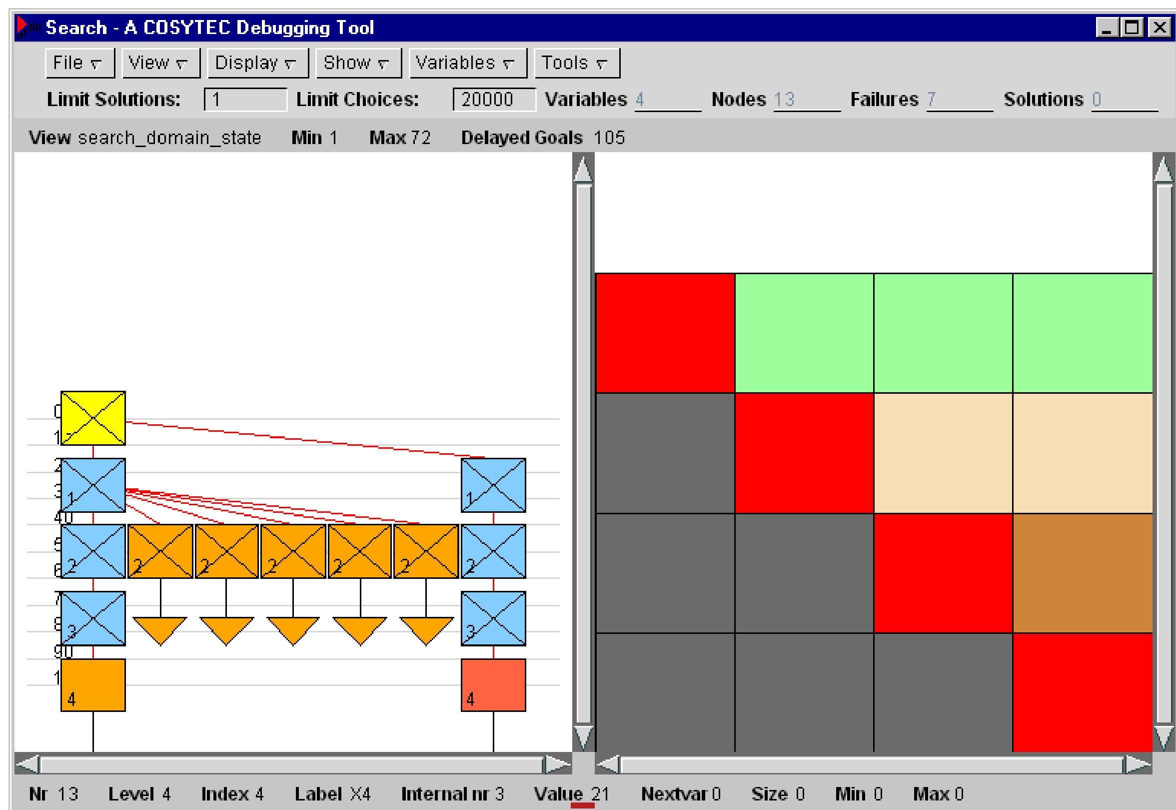


Figure 56: Combination

Finally, if we combine all the improvement techniques, we get this search tree. But this is really in the area of diminishing returns, as we do a lot of constraint reasoning in order to remove just a few nodes in the tree. The execution time of this program is well above the time required for the previous ones.

The program itself is also much more complex, as we not only have three models inside the program, but also the constraints which link the different models.

5.15 Summary

5.15.1 Cutting stock problem

- 0/1 model in finite domains rather poor
- finite domain model creates small search space
- linear model with branching even better

5.15.2 Improvements

- use of redundant linear model improves propagation
- minimize optimization reduces re-discovery
- labeling on cost: impact on overall cost
- domain splitting avoids immediate failures
- domain splitting not visible in search tree
- linear model more powerful than element model

6 Producer/Consumer

6.1 Problem

6.1.1 Introduction

We are considering a scheduling problem in the process industry. A single production line is running continuously to produce chemicals of different quality grades. The production is scheduled by switching from one grade to another at given time points. Customer orders are not handled individually, but are grouped together into a general demand, which is typically satisfied from stock. The production schedule must replenish the stock so that each material is always available. On the other hand, finished product storage is quite limited and is dedicated to the different product grades. The scheduling problem consists in defining start and duration of production runs for each product quality over a one-month period. Set-up constraints limit the possible sequences of products due to clean-out times and the wish to limit product down-grading. For each product grade, we can consider a producer consumer problem. An unknown number of production runs of unknown duration plus a given, initial stock must satisfy the demand at each time point, while never exceeding the limited storage capacity. A typical solution consists in making production runs as late as possible for each product, to avoid problems with the storage capacity. At the same time, the run length is set either to a typical value (often process dependent) or to the maximal length that the storage capacity allows. The need to co-ordinate the runs for different products, considering setup and individual demand, creates the challenge in this scheduling problem.

6.1.2 Problem Description

The problem described above is too complex for a detailed study. In the following we concentrate on the following sub problem. We consider a sequence of production runs for one product only. The product is made on a single, disjunctive machine, so that an order of the production runs can be given. We fix the number of runs a priori to a large number (possible runs). Only those runs before a cut-off date will be actually made, the others consist of dummy runs. A production run can last between 1 and 2880 minutes, with a preferred time of 1080 minutes. All quantities have been converted into production time, a production run of A minutes produces A units of product, which is available at the end of the run. The demand is given for fixed time points, all quantities are also expressed in production time requirements. We ignore the maximal stock level constraint for the moment, and only look at the minimal stock level constraint.

We use the following notation:

- **N** the number of producer events
- **M** the number of consumer events
- **L** the initial stock level
- $\mathbf{T_i}$ the end time of producer task i
- $\mathbf{P_i}$ the quantity produced in producer task i
- $\mathbf{S_j}$ the time point of consumer task j

- $\mathbf{C_j}$ the quantity consumed in consumer task j

We assume that producer and consumer tasks are ordered, i.e.

$T_i < T_{i+1}$

$S_j < S_{j+1}$

We must produce at least the quantity that we consume, but can produce more if we want to, so the following inequality holds:

$$\sum_{i=1}^{N} P_i + L \geq \sum_{j=1}^{M} C_j \quad (1)$$

At each time point, the accumulated production plus the initial stock must exceed the consumption up to this time point. We can express this condition with sets of disjunction between producer tasks and consumer tasks:

$$T_i > S_j \vee \sum_{k=1}^{i-1} P_k + L \geq \sum_{k=1}^{j} C_k$$ for all i in {1..N} and all j in {1..M} (2)

This condition states that the total production before consumer task j (which consists of all producer tasks 1..i-1 plus the initial stock) must exceed the total consumption (which consists of all consumer tasks 1..j).

We can further tighten the inequality constraints between the producer tasks by considering the temporal relation between the ends. If task i produces Pi units and finishes at time Ti, then its start time must be Ti-Pi. As the production is disjunctive, the previous task must have ended before that time point. We can thus add the constraints

$T_{i+1} \geq T_i + P_{i+1}$ for all i in {1,N-1}

Note that this formula is not the usual time relation between tasks, expressed between start times, but a relation between the end times of the tasks.

For the particular example studied here, the demand is given in the following table:

Time point	Quantity
2880	1440
7200	1440
12960	1440

The initial stock is given as 720 units, the planning horizon is 28800 units. Any task later than 20000 is considered a dummy run.

We are looking for a solution satisfying two criteria:

- Each run should be as late as possible
- Each run should have the preferred quantity 1080, except for dummy runs.

We consider as objective A the situation where preference is given to the lateness of the runs, and objective B where preference is given to the run size. Both problems have different solutions, as we will show below.

6.2 Model

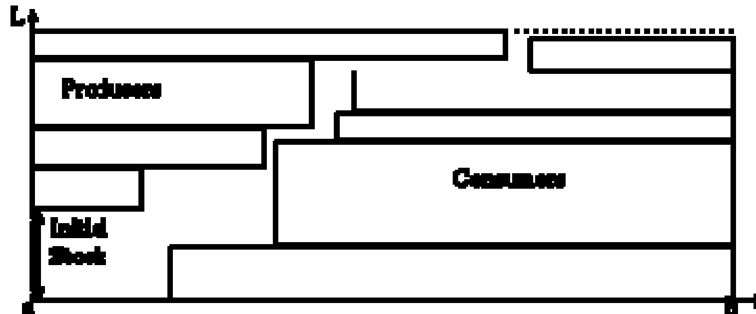


Figure 57: Producer Consumer Model

6.3 Program

```
top(Producer):-
        consumers(Consumer),
        producers(Producer),
        Initial_stock = 720,
        Final_stock :: 0..2880,
        End = 28800,
        producer_consumer(Producer,Consumer,Initial_stock,Final_stock,End),
        fix_values(Producer,1).

consumers([cons(2880,1440),cons(7200,1440),cons(12960,1440)]).

producers([prod(T1,Q1),prod(T2,Q2),prod(T3,Q3),prod(T4,Q4),prod(T5,Q5),

prod(T6,Q6),prod(T7,Q7),prod(T8,Q8),prod(T9,Q9),prod(T10,Q10)]):-
        [T1,T2,T3,T4,T5,T6,T7,T8,T9,T10] :: 0..28800,
        [Q1,Q2,Q3,Q4,Q5,Q6,Q7,Q8,Q9,Q10] :: 1..2880,
        T1 #= Q1,
        T2 #= T1+Q2,
        T3 #= T2+Q3,
        T4 #= T3+Q4,
        T5 #= T4+Q5,
        T6 #= T5+Q6,
        T7 #= T6+Q7,
        T8 #= T7+Q8,
        T9 #= T8+Q9,
        T10 #= T9+Q10.


producer_consumer(Producer,Consumer,Initial_stock,Final_stock,Final):-
        Limit :: 0..32000,
        producer_tasks(Producer,Start1,Dur1,Res1,End1,Initial_stock,Sum),
        consumer_tasks(Consumer,Final,Start2,Dur2,Res2,End2),
        append(Start1,Start2,Start),
        append(Dur1,Dur2,Dur),
        append(Res1,Res2,Res),
        append(End1,End2,End),
```

```
        Limit #<= Sum,
        cumulative(Start,Dur,Res,End,unused,Limit,Final,unused).

producer_tasks([],[],[],[],[],S,S).
producer_tasks([prod(T,Q)|P1],[0|S1],[T|D1],[Q|R1],[T|E1],Sum,Send):-
        producer_tasks(P1,S1,D1,R1,E1,Q+Sum,Send).

consumer_tasks([],_,[],[],[],[]).
consumer_tasks([cons(T,Q)|C1],Final,[T|S1],[D|D1],[Q|R1],[Final|E1]):-
        D :: 0..30000,
        T + D #= Final,
        consumer_tasks(C1,Final,S1,D1,R1,E1).

fix_values([],N).
fix_values([prod(T,Q)|R],N):-
        indomain(T,max),
        indomain(Q,1080),
        N1 is N+1,
        fix_values(R,N1).
```

6.4 Searchtree

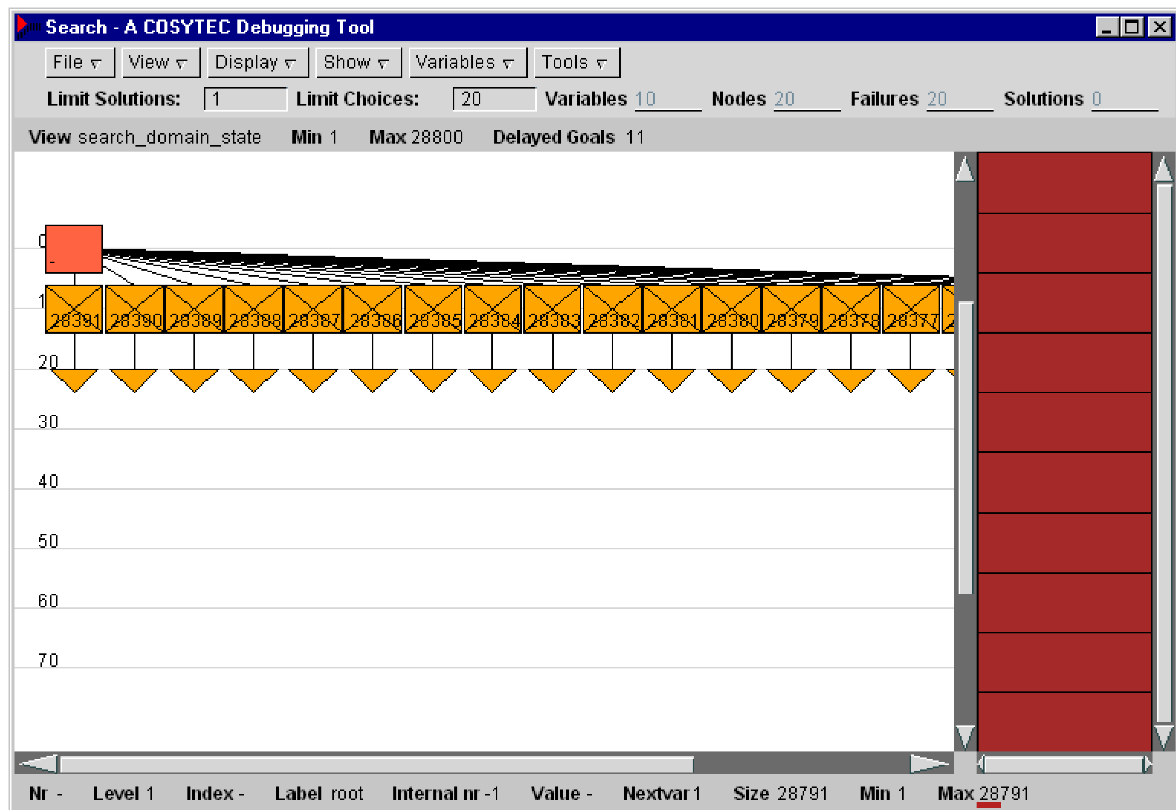


Figure 58: Searchtree

6.5 Analysis

6.5.1 Search tree

This is a total failure! The program assigns the first variable to a very high value, then fails on the second variable. And this goes on and on...

It is a good thing that we stop the searchtree tool after 20 choices, as the search would continue at this level until the first variable has reached value 2880.

You can also note that the enumeration is very slow, as all values for the second node are tested before the failure is detected and backtracking occurs.

6.5.2 Initial domains

The initial domains also are not restricted at all. As the duration of the tasks is not known, the ordering of the tasks and thus the impact of the first task on all other tasks is not visible.

6.5.3 Conclusion

This is about the worst type of result we can obtain. Clearly, the constraint model is not working at all. There are too many degrees of freedom and not enough propagation to restrict the choices.

6.6 Restrict

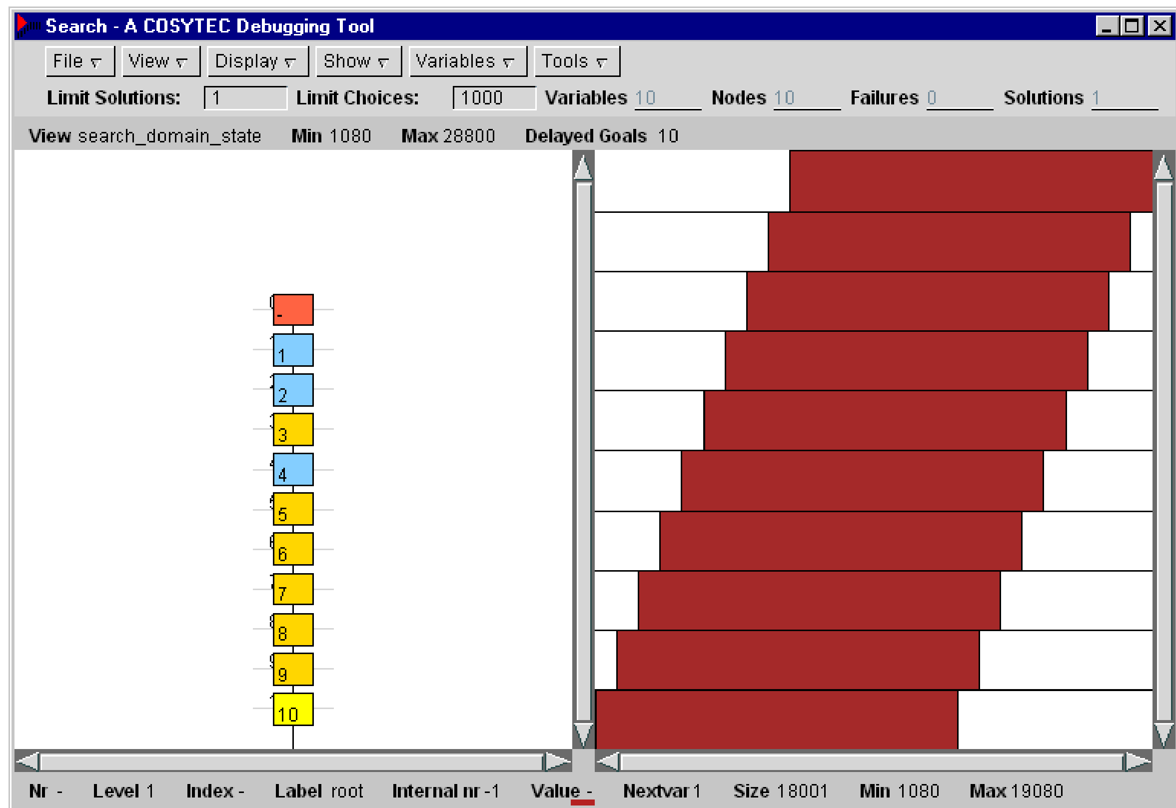


Figure 59: Restriction of problem

As a first modification, we can restrict the problem by enforcing the size of the producer tasks. If we want the tasks to have size 1080, we should perhaps force this size already in the domain definition. This will also fix the overall resource limit to an integer value. We change one line in the program:

```
[Q1,Q2,Q3,Q4,Q5,Q6,Q7,Q8,Q9,Q10] :: 1080..1080,
```

We now obtain a solution, instrumentation indicates zero backtracking steps and the search tree generated by the program (shown above) shows a straight line!

In each node on the left, we see the selected value for the variable. On the right we see the initial domain of all variables before starting the search, the root-node is selected in the tree.

6.7 Restrict analysis

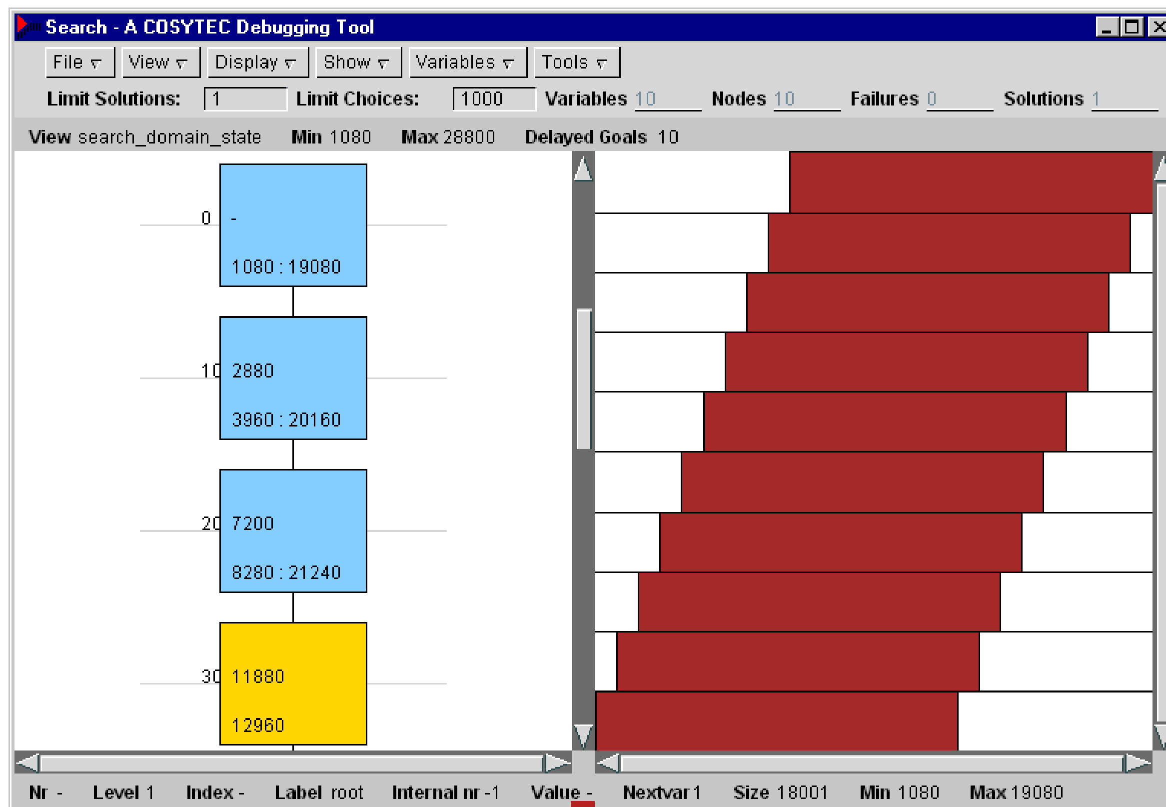


Figure 60: A closer look

All is well, isn't it? Not quite, since the program still runs for a rather long time. We can see what is happening by selecting the option 'Show domains' in the search tree tool.

For the first variable, the domain is 1080..19080, but the selected value is 2880. Remember that we use indomain(X, max) to assign values. This means that we have tested all values from 19080 downwards, before finding the consistent value 2880. Obviously, we spend a lot of time exploring these immediate failures. The constraint propagation is too weak and does not remove all inconsistent values.

It is important to note that this type of problem occurs with many constraint programs, but is rarely explained/mentioned in the result analysis. The classical backtracking count does not recognize this, neither does a simple search tree. Only looking at the domain values before and after the assignment we can find this cause of problems.

6.8 Alternative Model (1)

How can we improve these rather disappointing results? The first attempt is to re-organize the

cumulative constraint to improve the reasoning. In the original model, the producer tasks are created as long, flat rectangles. A restriction on one producer is only visible when we know the height of the other producers and their impact on the overall resource limit. With variable height of tasks, this impact is very weak and the resulting propagation not satisfactory. But we can re-organize the producer tasks in this problem, since we know that they are sorted in time. Instead of using wide, flat rectangles, we can use narrow, high rectangles as shown in the figure below:

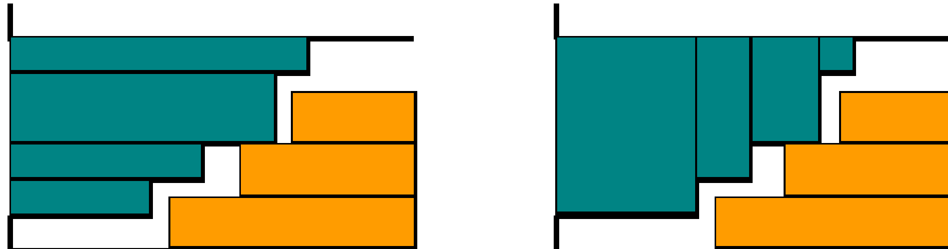


Figure 61: Alternative cumulative

Obviously, the area occupied by the producer tasks is the same, both formulations are equivalent. But given say the first task, we no longer need to know all other tasks to do some deduction.

Unfortunately, this improvement is not enough to handle the case of variable height tasks. Even with this modification, the program still continues to backtrack early.

If we fix the task height to 1080 by domain definition, the modified constraint reduces the domains a bit more, but still not sufficiently. Table 2 compares the propagation of programs producer1 and alternative1. It shows the domain of each variable just before its assignment and the first consistent value found. The highlighted fields show when the propagation has removed all inconsistent values above the first consistent one.

producer1	alternative 1	first value chosen
1080..19080	1080..2880	2880
3960..20160	3960..9000	7220
8280..21240	8280..11880	11880
12960	12960..21600	12960
14040..23400	14040..23400	23400
24480	24480	24480
25560	25560	25560
26640	26640	26640
27720	27720	27720
28800	28800	28800

6.9 Alternative searchtree

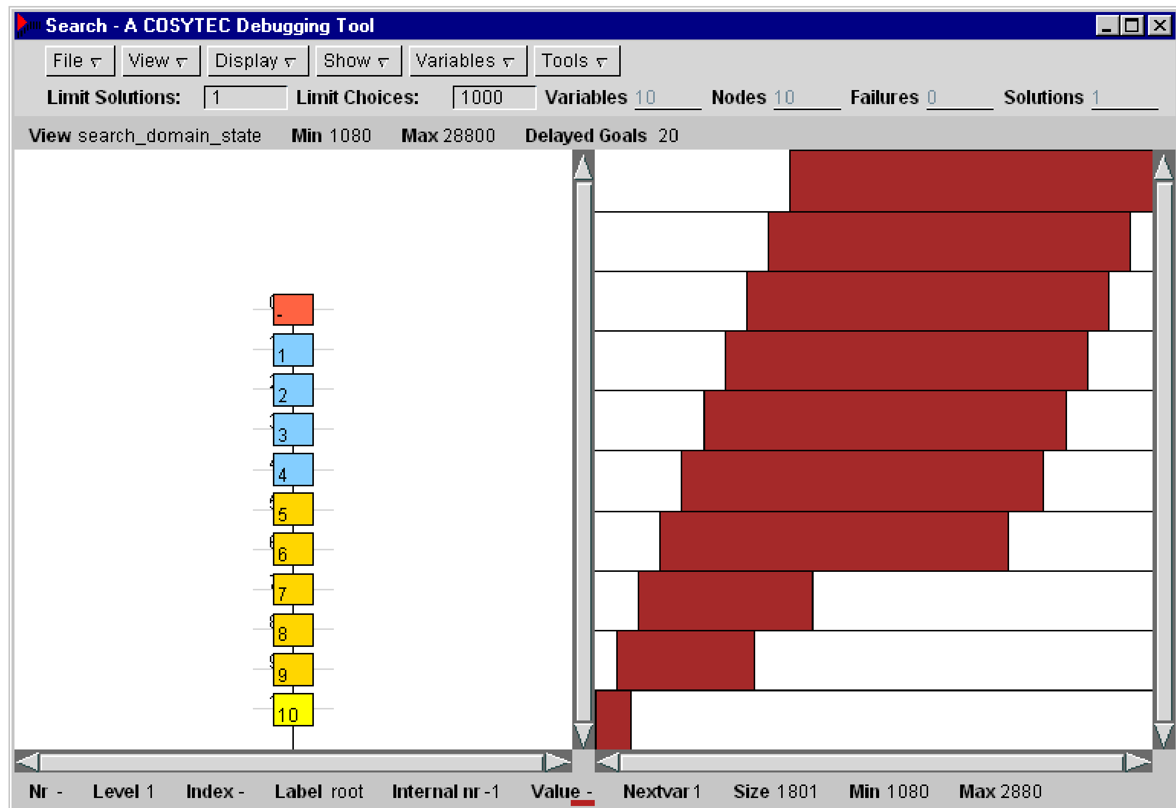


Figure 62: Alternative model (1)

6.10 Alternative Model (2)

While the program above shows some improvements for the fixed size case, it does not handle the variable height situation at all. What alternatives do we have?

Scanning through the CHIP reference manual, we find a possible candidate in the region parameter of the diffn constraint. The region parameter checks the utilization by tasks for a given region in an n-dimensional space. Different variants are possible, counting the difference between first and last utilization, the overall space used or the overall amount used in one dimension only. The parameter is normally used to restrict the amount of work that can be placed into a certain area, for example how much work can be performed by one machine in a given time period. We can (mis-)use this parameter to express that in the time before a consumer demand we want to producer at least the total consumption up to this point minus any initial stock. For each consumer task, we generate a new region with a new utilization domain variable, for which we constrain the minimum of the value. Each producer task is represented with a task of height one, duration equal to the quantity produced and start and end times according to the scheduled time. The figure below shows the layout:

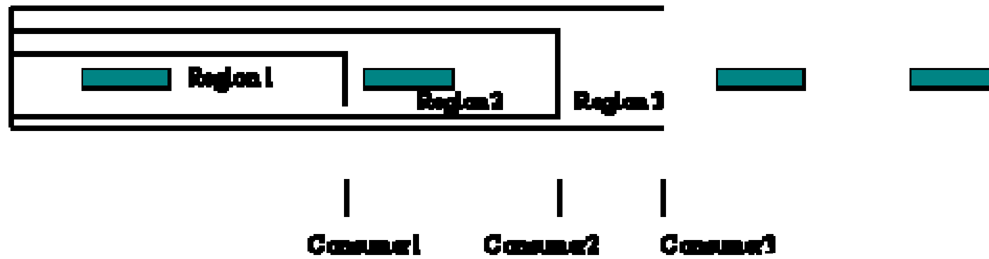


Figure 63: Region constraints

6.11 Program (2)

The program only shows the modified parts.

```
top(Producer):-
        consumers(Consumer),
        producers(Producer),
        Initial_stock = 720,
        Final_stock :: 0..2880,
        End = 28800,
        producer_region(Producer,Consumer,Initial_stock,Final_stock,End),
        fix_values(Producer,1).

producer_region(Producer,Consumer,Initial_stock,Final_stock,Final):-
        producer_region_tasks(Producer,Rectangles),
        producer_region_regions(Consumer,Initial_stock,Regions),
        diffn(Rectangles,unused,unused,unused,unused,Regions).

producer_region_tasks([],[]).
producer_region_tasks([prod(T,Q)|P1],[[X,Y,W,H]|R1]):-
        X :: 0..32000,
        T #= X+Q,
        Y = 0,
        W = Q,
        H = 1,
        producer_region_tasks(P1,R1).

producer_region_regions([],_,[]).
producer_region_regions([cons(T,Q)|C1],Stock,[[[0,0,T,1],-1,Use]|R1]):-
        Q1 is max(0,Q-Stock),
        Use :: Q1..T,
        producer_region_regions(C1,Stock,R1).
```

6.12 Alternative searchtree (2)

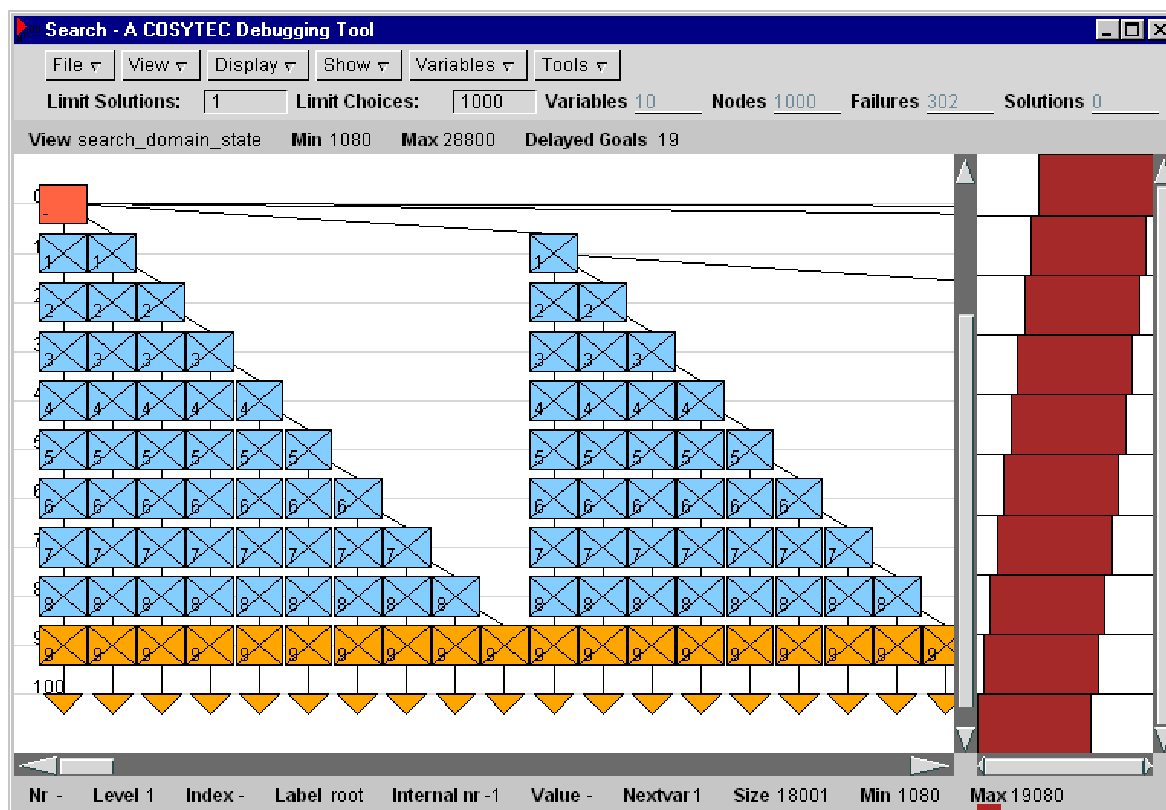


Figure 64: Alternative searchtree (2)

This is a complete and total failure, the constraint only updates the lower limit of the utilization once all tasks have been assigned. This indicates a missing propagation method in the diffn region constraint. The form of the search tree is typical for a very passive constraint. It always fails at the same level, the complete tree above the failure is generated and it does not find a solution within a reasonable time limit.

Well, perhaps in the next version of CHIP we will have the proper propagation rule...

The model itself is actually quite nice, as it allows to partially allocate some production to a consumer, if the producer task lies partially in the corresponding region.

Note that we can not invert the regions and state limits on the maximum utilization of time after the consumer event, as we do not know the total quantity that will be produced.

6.13 Alternative Model (3)

What next? Well we can use the diffn constraint for other things as well. In the original producer/consumer model we took figure 3 and treated it as a cumulative problem. But we can also see this as a 2D placement problem! The consumer tasks are fixed, so they correspond to fixed rectangles. The producer tasks must be stacked on top of each other. We can do this easily by linking the y-coordinates of the tasks with equality constraints. We again use here the fact that the temporal sequence of the tasks is given. In the general producer/consumer situation we could not express this rules so easily.

6.14 Program (3)

The program only shows the modified parts.

```
top(Producer):-
        consumers(Consumer),
        producers(Producer),
        Initial_stock = 720,
        Final_stock :: 0..2880,
        End = 28800,
        producer_placement(Producer,Consumer,Initial_stock,Final_stock,End)
,
        fix_values(Producer,1).

producer_placement(Producer,Consumer,Initial_stock,Final_stock,Final):-
        producer_rect(Producer,Initial_stock,Rect1),
        consumer_rect(Consumer,Final,0,Rect2),
        append(Rect1,Rect2,Rect),
        diffn(Rect).

consumer_rect([],_,_,[]).
consumer_rect([cons(T,Q)|C1],Final,Y,[[T,Y,W,Q]|R1]):-
        Y1 is Y+Q,
        W is Final-T,
        consumer_rect(C1,Final,Y1,R1).

producer_rect([],_,[]).
producer_rect([prod(T,Q)|P1],Level,[[0,Y,T,Q]|R1]):-
        Y :: 0..30000,
        Y #= Level,
        producer_rect(P1,Y+Q,R1).
```

6.15 Alternative searchtree (3)

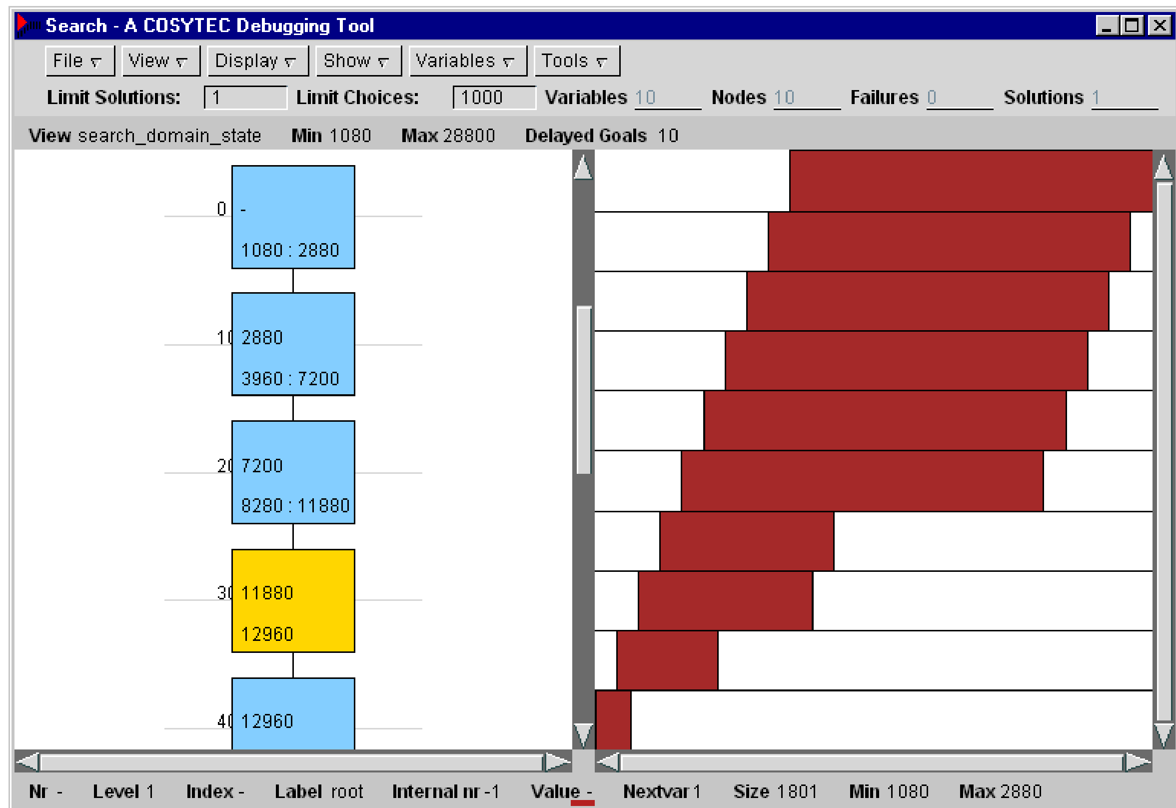


Figure 65: Alternative searchtree (3)

The tree on the left shows that the domain restrictions are now much better than before (compare to the table above), the initial domains on the right also show this improvement. In fact, for the case of fixed height this is the best possible result.

If we remove the restriction on the producer task size, then the program is not yet ideal. We can distinguish two situations.

- For each producer task, we assign first the start time (as late as possible) and then decide on the quantity. This finds a solution without backtracking, where task3 has size 1440, tasks 1 and 2 have size 1080.
- If we assign the quantity first and the fix the start, we still need a very long backtracking before finding the solution. For task3, after assigning size 1080, a wrong time is chosen (too late). There is not enough time before the third consumer task to make the missing quantity. Again, we have a lack of propagation here which forces us to enumerate many inconsistent values before finding the solution with a task4 of size 360.

6.16 Constraint Discovery

To further improve the program, we have to go back to its definition in formula (2). We can derive from this formula the following implication, which is a necessary condition for (2)

$$T_i > S_j \Rightarrow \sum P_k + L \geq \sum C_j$$ for all i in {1..N} and all j in {1..M} (3)

In this formula, we can replace the time needed to produce task i with the time that is left after producing task i-1, but before the consumer task j. The following implication still holds:

$$T_i < S_j \Rightarrow \sum P_k + L + S_j - T_i \geq \sum C_j$$ for all i in {1..N} and all j in {1..M} (4)

In CHIP, we can add this implication with conditional propagation in the form of a if-then-else constraint.

6.17 Program (4)

```
top(Producer):-
        consumers(Consumer),
        producers(Producer),
        Initial_stock = 720,
        Final_stock :: 0..2880,
        End = 28800,
        producer_placement(Producer,Consumer,Initial_stock,Final_stock,End)
,
        producer_generate(Producer,Consumer,Initial_stock,0),
        fix_values(Producer,1).


producer_generate(P,[],_,_).
producer_generate(P,[cons(S,C)|C1],L,Made):-
        producer_generate1(P,S,C,L,Made+C),
        producer_generate(P,C1,L,Made+C).

producer_generate1([],S,C,L,Made).
producer_generate1([prod(T,Q)|P],S,C,L,Made):-
        if T # S then L #= Made,
        if T #= Made+T,
        producer_generate1(P,S,C,L+Q,Made).
```

6.17.1 Alternative searchtree (4)

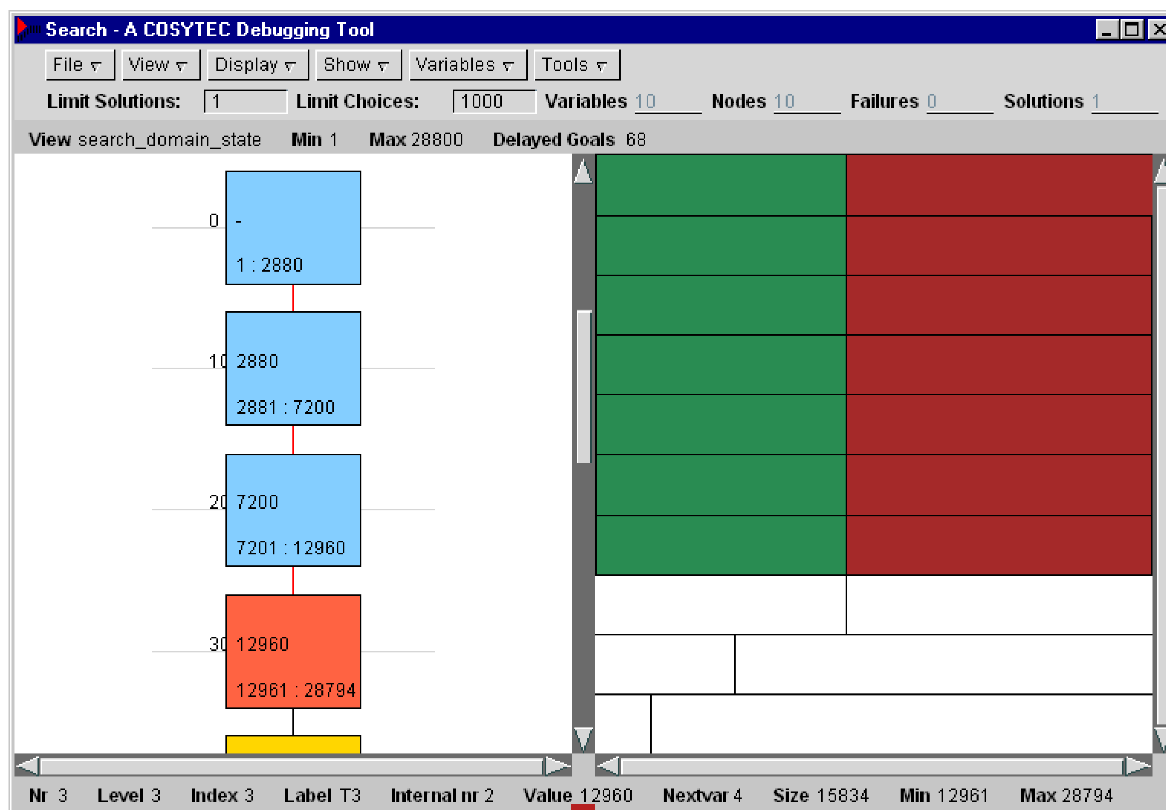


Figure 66: Constraint discovery

This addition solves the problem. We find solutions for both objectives A and B without backtracking, just by propagation without testing inconsistent values.

The last modification is to remove the diffn constraint and to keep the implications only. This lands us back at square one. There is not enough propagation, we start enumerating inconsistent values. As we have replaced disjunction (2) with an implication, this is not surprising. We only get the propagation in one direction. In order to solve the complete problem by inequalities, we would have to introduce additional 0/1 variables to model the disjunction.

6.18 Summary

We have considered a number of alternative models for the producer/consumer constraint where tasks are ordered, but number and size of the tasks are not determined a priori. What lessons we can learn from this experience?

6.18.1 Standard model not sufficient

The standard model using cumulative was not sufficient to solve even a simple example. The uncertainty about the domain variables is too big to perform good propagation. Changing the model from horizontal to vertical producer tasks improves the propagation, but not enough.

6.18.2 Region model not useful

The region constraint in diffn does not provide constraint propagation in the required direction. It is intended to restrict the maximum usage in a region, not to enforce a minimum usage as required here. The trick of inverting the region and constraining everything outside it does not work as the total usage is not known a priori.

6.18.3 Placement model ok, but not perfect

The 2D placement model of the producer/consumer constraint requires the fact that the tasks are ordered in time. It performs much better than the standard model, but not well enough. If we add some necessary implications, we finally obtain very good propagation.

6.18.4 No backtracking is not enough

As we could see on program producer1, it is not enough to have zero backtracks in a solution . We must also consider the effort required to remove inconsistent values, which immediately fail after the assignment. The search tree tool can be used to discover and understand such problems.

6.18.5 Use of search tree tool

The search tree tool and the global constraint visualization can provide useful hints to understand what is happening in a program. They also provide a simple way to explain problems to others without writing expensive ad-hoc programs.

6.18.6 Think small

It is always a good idea to isolate a small part of a larger application problem in order to understand the basic propagation mechanism. It would be quite difficult to directly understand the producer/consumer constraint behavior within the context of a large scale application with multiple production lines, dozens of products and all kinds of other constraints. Once the basic constraint behavior is understood, we can much more easily understand what is happening in the larger application.

7 Acknowledgment

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

8 Bibliography

[AB93] 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

[BBC97] N. BELDICEANU, E. BOURREAU, P. CHAN, D. RIVREAU, Partial Search Strategy in CHIP , *2nd Int Conf on Meta-heuristics*, Sophia-Antipolis, France, July 1997

[BC94] N. BELDICEANU, E. CONTEJEAN, Introducing Global Constraints in CHIP, *Journal of Mathematical and Computer Modelling*, Vol 20, No 12, pp 97-123, 1994

[BDD97] F. BUENO, P. DERANSART, W. DRABENT, G. FERRAND, M. HERMENEGILDO, J. MALUNSZYNSKI AND G. PUEBLA, On the Role of Semantic Approximations in Validation and Diagnosis of Constraint Logic Programs. *In Proc. Of the 3rd. Int'l Workshop on Automated Debugging-AADEBUG'97*, Pages 155-170, Linköping, Sweden, May 1997.

[CGH93] M. CARRO, L. GOMEZ, M. HERMENEGILDO, Some Paradigms for Visualizing Parallel Execution of Logic Programs, *Proc. ICLP93*, Budapest, Hungary. MIT Press, Cambridge, MA, 1993

[FAB97] M. FABRIS ET AL., CP Debugging Needs and Tools, In *Proc. Of the 3rd. Int'l Workshop on Automated Debugging-AADEBUG'97*, Pages 103-122, Linköping, Sweden, May 1997.

[HK71] M. HELD, R. KARP, The Travelling Salesman Problem and Minimum Spanning Trees: Part II, *Mathematical Programming* 1(1971), pp 6-25

[JM94] J. JAFFAR AND M.J. MAHER, Constraint Logic Programming : A Survey. *Journal of Logic Programming*, 19/20:503-581, 1994.

[JON96] C. V. JONES, *Visualization and Optimization*, Kluwer Academic Publishers, Norwell, USA, 1996

[MEI95] 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.

[SCH97] 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.

[SIM95] H. SIMONIS, Application Development with the CHIP System*, Proc Contessa Workshop*, Friedrichshafen, Germany, September 1995, Springer LNCS

[SIM95A] H. SIMONIS, The CHIP System and its Applications, *Proc. Principles and Practice of Constraint Programming*, Cassis, France, September 1995

[SIM96] H. SIMONIS, A Problem Classification Scheme for Finite Domain Constraint Solving, *Proc workshop on constraint applications*, CP96,Boston, August 1996

[WAL95] M. WALLACE, Survey: Practical Applications of Constraint Programming, *Constraints*, Vol. 1, Nr1-2, pp 139-168, September 1996