Constraint-Based Filtering and Evaluation of CSP Search Trees

Maximilian Bels, Sven Löffler and Ilja Becker and Petra Hofstedt
Programming Languages and Compilers Group, Brandenburg University of Technology, Konrad-Wachsmann-Allee 5, Cottbus, Germany

Keywords: Constraint Programming, Finite-Domain Constraint Satisfaction Problem, CSP, Search Tree, Constraint-Based Filtering.

Abstract: Using Constraint Programming (CP) real world problems can be described conveniently in a declarative way with constraints in a so-called constraint satisfaction problem (CSP). Finite domain CSPs (FD-CSPs) are one form of CSPs, where the domains of the variables are finite. Such FD-CSPs are mostly evaluated by a search nested with propagation, where the search process can be represented by search trees. Since search can quickly become very time-consuming, especially with large variable domains (solving CSPs is NP-hard in general), heuristics are used to control the search, which in many cases — depending on the problem — allow to achieve a performance gain. In this paper, we present a new method for filtering and evaluating search trees of FD-CSPs. Our new tree filtering method is based on the idea of formulating and evaluating filters as constraints over FD-CSP search trees. The constraint-based formulation of filter criteria proves to be very flexible. Our new technique was integrated into the Visual Constraint Solver (VCS) tool, which allows the solution process of CSPs to be followed interactively and step by step through a suitable visualization.

1 INTRODUCTION

Finite-Domain Constraint Satisfaction Problems (FD-CSPs) are evaluated by search nested with propagation. Search heuristics control the search. They affect the structure of the search trees of CSPs and can, thus, influence the performance of CSP evaluation. Our aim is to compare, review and better understand the impact of search heuristics.

Search trees can quickly become extraordinarily large. At the same time, for a comparison or review, one only wants to consider certain sections of the tree. Thus, we want to be able to formulate and evaluate criteria for particularly interesting sets of search nodes and parts of search trees.

In this paper, we present a new method for formulating criteria and filtering and evaluating search trees of FD-CSPs. The main idea is to formulate the search tree filters themselves as CSPs. Data of nodes and their relations in subtrees can be described very flexibly by variables and constraints. A constraint solver can be used to evaluate the filter CSPs and provide optimized solutions.

Our new method was integrated into the Visual Constraint Solver (VCS). VCS is a tool for the visualization of FD-CSP evaluation developed at the Programming Languages and Compilers group (PSCB) at BTU Cottbus-Senftenberg. VCS takes as input MiniZinc (MiniZinc, 2022) programs describing CSPs and allows to visualize the static CSP as well as the solution process by a search tree and log files.

Related Work. Some tools for the visualization of CSPs and search trees have been documented in the literature, among them a constraint graph viewer for the platform G12 (Mak, 2022), DPViz (for SAT problems) (Sinz and Dieringer, 2005), CPViz (Simonis et al., 2010) and DrawCSP (Li and Epstein, 2010). These tools are typically designed for specific constraint languages or solvers, they often visualize CSP networks, but do not visualize the search trees, and even if so, they do not allow to define filters on trees as flexible as our method does. In contrast, VCS works on MiniZinc models which are solver independent, is very flexible in the definition of search tree filters, supports the interactive visualization of search trees and node properties and even allows to compare different search strategies at the same time. Moreover, VCS supports the visualization of (multi-dimensional) arrays, which is a further unique feature of our tool.

The paper is structured as follows: Section 2 recalls basic terms and definitions of the area of con-

Bels, M., Löffler, S., Becker, I. and Hofstedt, P.
Constraint-Based Filtering and Evaluation of CSP Search Trees.
DOI: 10.5220/0011641100003393
In *Proceedings of the 15th International Conference on Agents and Artificial Intelligence (ICAART 2023) - Volume 3*, pages 220-227
ISBN: 978-989-758-623-1; ISSN: 2184-433X

straint programming. We introduce the notions of filter criteria and filter models in Section 3. Following, Section 4 discusses technical and implementation details of our constraint-based approach to filtering search trees as well as the visualization inside our tool VCS. Section 5 demonstrates the modelling of further useful filters. Finally, in Section 6 we conclude the paper and give directions of future research.

2 CONSTRAINT PROGRAMMING

This section provides basic concepts and definitions of constraint programming (based on (Dechter, 2003)).

Definition 1 (constraint)**.** *Let X be a set of variables. A* constraint *$c = (X', R)$ is a relation R over a subset X' of the variables of X, i.e. $X' \subseteq X$.*

The relation R of a constraint $c = (X, R)$ represents a subset of the Cartesian product of the domain values $D_1 \times ... \times D_n$ of the corresponding variables $x_1, ..., x_n \in X$. It can be given explicitly by the concerning value tuples or implicitly by a mathematical description.

Definition 2 (CSP)**.** *A* constraint satisfaction problem (CSP) *is defined as a triple $P = (X, D, C)$, where*

- *$X = \{x_1, \ldots, x_n\}$ is a set of variables,*
- *$D = \{D_1, \ldots, D_n\}$ is a corresponding set of domains, i.e. D_i is the domain of x_i, and*
- *$C = \{c_1, \ldots, c_m\}$ is a set of constraints.*

In the following, we only consider *finite domain-CSPs (FD-CSPs)* (and write CSP), where the variable domains are finite.

Example 1. The n-queens problem aims to place n queens on an $n \times n$-chess board such that the queens do not attack each other. For example, for the 5-queens instance we can give a CSP $P = (X, D, C)$ as follows:

- $X = \{q_1, \ldots, q_5\}$ are variables to represent the queens, such that queen q_i is placed on column i,
- $D = \{D_1, \ldots, D_5 | D_1 = \ldots = D_5 = \{1,2,3,4,5\}\}$ are the domains of the variables q_i; they represent the row numbers on the board,
- $C = \{$*alldifferent*$(\{q_1, \ldots, q_5\}),$ *alldifferent*$(\{q_i + i \mid i \in \{1, \ldots, n\}\}),$ *alldifferent*$(\{q_i - i \mid i \in \{1, \ldots, n\}\})\}$ are the constraints to ensure that the queens cannot threaten each other. Here, the first constraint says that all q_i should be different, this ensures distinct rows for the queens. The other two constraints determine distinct diagonals. At this, the *alldifferent* constraint is a so-called *global* constraint which ensures that the values of all included variables are distinct.

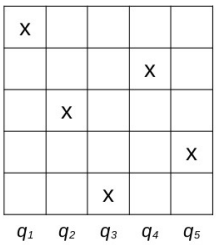

Figure 1: A solution of the 5-queens problem.

A *solution* of a CSP is an instantiation of all its variables with values from their domains, that satisfies all the constraints (Dechter, 2003). For example, the 5-queens problem has 10 solutions, including e.g. $q_1 = 1,\ q_2 = 3, q_3 = 5, q_4 = 2, q_5 = 4$ as illustrated by Figure 1.

The search for solutions of a CSPs is realized by constraint solvers which use backtracking-based depth-first search. To speed-up the search, it is interleaved with constraint propagation steps. Each individual constraint describes a set of allowed tuples. By propagating a constraint one can constrain the search space locally, i.e. values that do not satisfy a constraint and are therefore not involved in any solution, are removed from the search space. The solver performs such propagation steps in alternation with variable instantiation during search. This process is described in detail e.g. in (Dechter, 2003; Marriott and Stuckey, 1998).

Furthermore, search heuristics are used in the search for solutions of CSPs. We distinguish between variable ordering and value ordering heuristics (van Beek, 2006).

Variable ordering heuristics decide which variable is next instantiated to a value during the backtracking-based search. Different variable heuristics typically yield differently structured search trees. The goal of these heuristics are strong domain restrictions early on and narrow search trees. This can be achieved e.g. with the *first-fail* heuristic, where variables with the smallest domains are chosen first. With a similar intention, the *most-constraint* heuristic first takes variables that are attached to many constraints of the given CSP.

Value ordering heuristics decide which value a previously selected variable is assigned. Value heuristics do not affect the general structure of the search tree, but they lead to a reordering of the sub trees of a search tree. This is particularly interesting and can

be advantageous if only a first solution or best solution(s) are sought or when parallel search or no-good learning is applied.

Both kinds of heuristics manipulate the search tree and thus can significantly affect the time required in the solution process. For this reason, it is important to evaluate different search heuristics in order to predict a good or potentially best strategy for solving a given problem as quickly as possible.

Constraint solvers, such as the Choco solver (Choco, 2022), provide a number of variable and value ordering heuristics. However, the appropriate choice of the heuristics is the responsibility of the user.

3 DEFINING FILTER MODELS

Our aim is to review and evaluate the performance of search heuristics by assessing and comparing search trees of CSPs. This is realized by filters on search tree node sets, expressed themselves as CSPs and evaluated by a constraint solver.

In this section, we first recall search trees. Next, we explain the notions and usages of properties of search tree nodes and filter criteria on sets of such nodes. On top of these, we define the notion of a filter model which allows to express constraints on node sets and subtrees of search trees.

3.1 Search Trees

Definition 3 (search tree)**.** *Let a CSP $P = (X,D,C)$ be given. A* search tree *for P is a tree T, whose nodes represent P enriched by instantiations to subsets of X. For every node in T, we assume that consistency enforcement has been applied. The root of T stands for P (after consistency enforcement and with no further variable instantiation). A child of a node n is an extension of the instantiation of n for exactly one variable v of X, where the size of the remaining domain of v is greater than one. A leaf is either a solution of P or inconsistent (noticed by $fail$).*

Example 2. We consider a CSP $P = (X,D,C)$ with $X = \{x,y,z\}$, $D = \{D_x = D_y = \{0,1,2\}, D_z = \{0,1,2,3\}\}$ and $C = \{x < y, z = x+y\}$. A search tree for P is given in Figure 2.

The root node stands for the original problem P, where propagation of the constraints has already been performed and, thus, the domains of the variables have been narrowed. E.g. from $x < y$ follows, that x cannot be 2 and y cannot be 0.

P has two child nodes P_1 and P_2. P_1 represents P, where we assigned 0 to x. These instantiations are noticed on the edges between the parent and child nodes. By further propagation for consistency enforcement, also the domains of the other variables may be affected, for P_1 e.g. 3 is removed from the domain of z.

Finally, for P the solution process reveals three solutions as shown in the leaf nodes of the tree.

3.2 Node Properties and Filter Criteria

To describe, to filter, and to cut out parts of a search tree with certain interesting properties with respect to the solution process we use properties of the search tree nodes and call them *node properties*. Since the search tree is dynamically generated resp. traversed during the solution process by backtracking search, node properties are gained dynamically during search.

We identified certain relevant node properties. At this, we take as basis a left-to-right backtrack search and consider properties which appear caused by the course of the search and the tree structure (properties a-i) as well as properties of the CSP and variable instantiations at the corresponding node (properties j-k). The node properties of node n include the following (this list is not exclusive and can even be extended by the user):

a the discovery index $n_{id} \in \mathbb{N}$ of the node n in the left-to-right backtracking order of the tree (the root node has discovery index 1),

b the number $n_{solutions} \in \mathbb{N}$ of solutions found so far (during backtracking search, including n), when the node n is reached in the tree,

c the number $n_{leafs} \in \mathbb{N}$ of leaf nodes (solutions and dead ends, including n) found so far during search,

d the depth $n_{depth} \in \mathbb{N}$ of the node in the search tree,

e the index $n_{parentId} \in \mathbb{N} \cup \{-1\}$ of the parent of node n (or -1 in case n is the root),

f a truth value (0 for false, 1 for true) n_{isLeft} to express, whether n is a left-most child of it's parent (or 1 in case n is the root),

g the index $n_{lastLeftParentId} \in \mathbb{N}$ of the closest left ancestor of n, that is, the closest ancestor where n_{isLeft} is 1,

h a truth value (0 for false, 1 for true) $n_{isSolution}$ to express, whether n represents a solution (leaf node),

i a truth value (0 for false, 1 for true) $n_{isContradiction}$ to express, whether n is a contradiction (leaf node),

j the size $n_{a,domainSize} \in \mathbb{N}$ of the domain of a certain variable a,

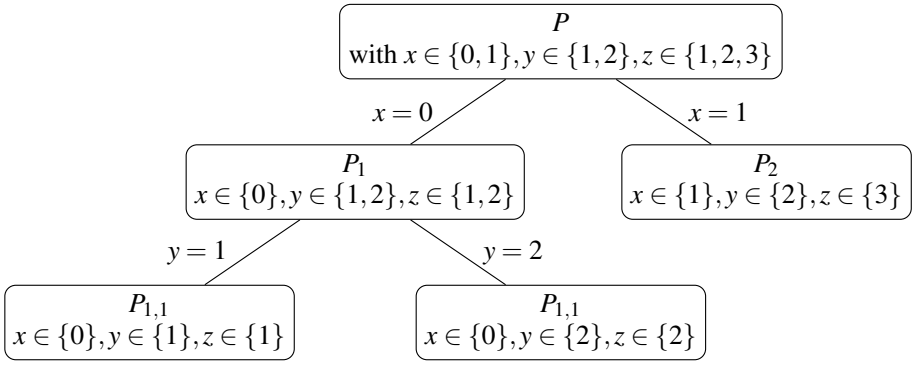

Figure 2: Search tree for Example 2.

k the domain value of a variable a, in case that the variable is finally instantiated $n_{a,instantiatedTo} \in D_a$ (otherwise undefined).

To analyse, to visualize, and to compare search trees to better understand search behaviour we want to filter search trees and nodes by certain criteria. We use the above explained node properties, either directly or for the formulation of complexer constraints over several nodes or parts of a search tree.

We give typical definitions of potentially interesting *filter criteria* for node sets and subtrees. Later on, we will see that the user of our system will be able to define such criteria and constraints by himself.

A *Variable instantiations.* A first, simple filter goal is to find the positions or nodes in a search tree, where a specific variable a is assigned a specific value v_a. Find a node n, such that holds: $n_{a,instantiatedTo} = v_a$.

B *Number of nodes visited to find a certain number of solutions.* As well, one can be interested to find out, how many nodes have been visited until m solutions were found. Determine a node n, such that holds: $n_{solutions} = m$. The discovery index n_{id} of n is the searched number of visited nodes.

There are also clearly more complex descriptions possible, which refer to subtrees or node sets, e.g. solution density or domain reduction.

C *Solution density.* The solution density describes the proportion of solutions in the set of leafs in a certain region of the search tree.

Definition 4 (solution density)**.** *The* solution density $den(n_1,n_2)$ *of a set of all nodes visited during search between the nodes* n_1 *(visited first) and* n_2 *(visited last) is defined by:*

$$den(n_1,n_2) := \frac{n_{2_{solutions}} - n_{1_{solutions}}}{n_{2_{leafs}} - n_{1_{leafs}}},$$

whereby $n_{1_{solutions}} \leq n_{2_{solutions}}$, $n_{1_{leafs}} \leq n_{2_{leafs}}$.

With additional constraints on the nodes, one can search for appropriate nodes n_1 and n_2 which determine a search tree. In this way, one can search for subtrees with a high/low solution density which may indicate areas in the tree, where the used heuristics are particularly effective/ineffective.

D *Domain reduction.* The domain reduction describes the strength of the reduction of a variable domain in the progression of the search between two nodes in the tree.

Definition 5 (domain reduction)**.** *The* domain reduction $red(n_1,n_2,a)$ *by search during tree traversal between the nodes* n_1 *and* n_2 *and with respect to variable* a *is defined by:*

$$red(n_1,n_2,a) := 1 - \frac{n_{2_{a,domainSize}}}{n_{1_{a,domainSize}}}.$$

At this, we assume, that n_2 *is a node in a subtree with root* n_1*. Thus, also* $n_{2_{a,domainSize}} \leq n_{1_{a,domainSize}}$ *holds.*

With domain reduction we can observe the influence of a heuristics on the domain of certain variables in search trees, e.g. we can filter for parts, where the domain is reduced by e.g. 50 %. With additional constraints on the indices of the two nodes n_1 and n_2 we can find positions in the tree with a fast/slow domain reduction for certain variables.

Again, and as for the filter properties holds: The set of filter criteria is extendable and programmable by the user of our system.

3.3 Filter Models

Depending on the application case, one may be interested in different node properties and evaluation criteria. E.g. consider a lecture in Constraint Programming, where the instructor wants to explain the idea of

search and search trees in general. In this case, it may be sensitive to focus on simple properties like node indexes, node depth in a search tree, or the number of decisions taken during search up to a certain node. In contrast, if one wants to optimize a search heuristics for a certain problem class, more complex, composed criteria (as e.g. in Definition 4) might be important.

To allow a definition of criteria as individually and freely as possible, we decided to define and handle themselves as CSPs. Such CSP models of criteria are called *filter CSPs* or *filter models* in the following.

A *filter model* is a CSP F modeled around an original CSP P (i.e. the *problem* CSP) and its search tree.

One the one hand, a filter model F may contain additional variables and constraints to enhance the original CSP P. This may help to shrink a given problem to be able to consider partial aspects of P, e.g. consider P, where a certain variable instantiation is somehow fixed. On the other hand (and more interestingly), the filter CSP F describes constraints over the nodes of the search tree. Thus, the variables of F represent tree nodes and tree node properties. Accordingly, the domain D_n of such a search node variable n is the set of all search tree nodes (represented by a unique node index), the domain $D_{n,prop}$ of a property variable of a tree node is the set of the possible values of such a property. Finally, constraints of F describe search node sets and constraints on them.

Definition 6 (filter model)**.** *Given a problem CSP* $P = (X,D,C)$*. A* filter CSP *(or* filter model*)* $F = (X_P \cup X_N, D_P \cup D_N, C_F)$ *consists of*

- *a set* X_P *of variables with finite domains* D_P *(to enhance the original CSP P),*
- *a set* X_N *of variables with finite domains* D_N *(to describe search tree nodes and search tree node properties)*
- *a set* C_F *of constraints over* $X_P \cup X_N \cup X$*.*

While the set C_F of constraints may contain further model constraints on P (as explained above), it primarily consists of constraints on search tree nodes. These constraints are used to specify node sets of a search tree by specifying desired node properties and relations between the nodes and node properties.

Example 3. Let the CSP P from Example 1 describing the 5-queens problem be given. A filter model $F = (X_N, D_N, C_F)$ with

- $X_N = \{n, n_{isSolution}\}$,
- $D_N = \{D_n, D_{n_{isSolution}} = \mathcal{B}\}$, where D_n is the set of all search tree nodes (represented by unique indexes), and
- $C_F = \{n \leq 30, n_{isSolution} = true\}$

specifies a filter which provides all solutions within the first 30 search tree nodes visited during backtracking search.

More complex examples, including the search for subtrees with restrictions on the domain reduction and solution density, are given in Section 5.

Since the original problem CSP P is an FD-CSP, it is ensured that all search trees of P are finite. Thus, the number of search tree nodes and their concerning properties is finite as well. However, in practice, we do not know in advance, the concrete nodes of a proper search tree nor their property values because a search tree is generated dynamicall during the evaluation of the original problem CSP P. Thus, while we are able to formulate the needed variables and constraints on them for the filter CSP F, we cannot provide their property values in advance. In this way, the filter CSP F is initially *incomplete*. During search additional constraints for defining the property values of the nodes must be added to F. This is discussed in more detail in the following section.

4 FILTER IMPLEMENTATION

The Visual Constraint Solver (VCS) is a tool for the visualization of FD-CSPs and their evaluation, developed at BTU Cottbus-Senftenberg in a series of bachelor/master theses (Buckenauer, 2019; Reda, 2020; Bels, 2022). As input, VCS takes MiniZinc programs which specify CSPs. MiniZinc (Stuckey et al., 2020; MiniZinc, 2022) is a high-level, solver-independent constraint modeling language whose constraint models are compiled via the intermediate language FlatZinc into other constraint programming languages such as Choco (Choco, 2022). Choco is a Java library for constraint solving and it is used inside VCS to solve FD-CSPs.

Filter models (as described in Section 3) are our new extension of VCS. Observe, that they are just CSPs on search tree nodes and their properties. Thus, the main idea of their implementation is to represent them by constraint models and to handle them by constraint solvers. This yields an extended workflow:

Filter models are defined just like problem CSP models using MiniZinc. They are as well compiled into a Choco constraint model F_{choco} via FlatZinc. When the original problem CSP P_{choco} is solved by the Choco solver, additional information about the nodes and node properties is collected and used to extend the Choco filter model. The resulting enhanced filter constraint model $F+_{choco}$ is again solved by the Choco solver. Its solutions are used to enhance and decorate the visualization of the solutions of P_{choco} in

```
1  include "alldifferent.mzn";
2  int: n = 5;
3  % n is the number of queens
4  array [1..n] of var 1..n: q;
5  % queen in column i is in row q[i]

6  constraint alldifferent(q);
7  % all queens in distinct rows and
       ...
8  constraint alldifferent([q[i] + i |
        i in 1..n]);
9  constraint alldifferent([q[i] - i |
        i in 1..n]);
10 % ... distinct diagonals
11 solve satisfy;
```

Listing 1: A MiniZinc model of the 5-queens problem.

```
1  var int: n_id;
2  var int: n_isSolution;
3  constraint n_id <= 30;
4  constraint n_isSolution = 1;
5  solve satisfy;
```

Listing 2: A filter for all solutions within the first 30 nodes visited during backtracking search.

the VCS tool by highlighting node sets satisfying the filter CSP.[1]

Listing 1 gives an example of a MiniZinc CSP model. It specifies the 5-queens problem CSP $P = (X, D, C)$ from Example 1. After an import of the global `alldifferent` constraint from the MiniZinc libraries (Line 1), the variables `q[1]` to `q[5]` (organized in an array) with their domain values `1` to `5` (Lines 2,4) are defined. In Lines 6-9 the constraints of set C are given as above. Line 11 initiates the solution process.

The script in Listing 2 is an example of a filter model in MiniZinc which corresponds to Example 3. It declares a search node `n` with unique identifier `n_id` and property `n_isSolution`. The constraints ensure that the desired node is a solution (`n_isSolution = 1`) and is visited within the first 30 nodes (`n_id <= 30`). The last line instructs the solver to search for a solution that satisfies all constraints. Notice, that properties must always be given in conjunction with a filter node index and may require to specify an associated variable in addition to that (here `n_isSolution`).

Visualization of Filter Results. When assessing a CSP and its solution process using VCS the user can load problem CSPs P and filter models F. Besides he can choose between variable and value ordering heuristics provided by Choco. The problem CSP can be solved step-wise and filter models can be applied interleaved.

Figure 3 shows a cut-out of the search tree of the 5-queens CSP from Example 1 and Listing 1 in VCS. We used the filter given in Listing 3 (to be explained in Section 5 in more detail). This filter model identifies parts of the tree with a solution density (cf. Definition 4) of at least 50%.

In the shown case, the search tree nodes with indices 16-22 include 4 leafs, where 2 of them (nodes 17 and 19) are solutions (solution density is 50%).

Green nodes represent solutions of the problem CSP P, red nodes stand for failing subtrees. Yellow coloured nodes represent a filter result. It is possible to switch step-by-step between the several filter results, here subtrees satisfying the filter model. The user of VCS can assess the filtered node data by clicking on a search tree node. Then a window opens and shows the node properties, e.g. at node 17 we see a solution of the 5-queens problem. (Notice, the `X_INTRODUCED_i` variable names result from the FlatZinc conversion of the queens array inside VCS and stand for the queens q_{i+1}, i.e. `X_INTRODUCED_0` for q_1, `X_INTRODUCED_1` for q_2 and so on).

5 APPLICATION EXAMPLES

Now, let us consider further examples of filter scripts which shall help to asses search heuristics. They are written in MiniZinc as before and provided as input for VCS together with a problem CSP P.

Listing 3 shows a filter specification for subtrees with a high solution density according to Definition 4. In Lines 1 and 6, two search tree nodes `n1` and `n2` are declared. Node `n1` is the root of the to be filtered subtree, node `n2` the subtree's rightmost leaf. For the actual definition of the filter criterion in Line 20, we need certain node properties gained with the code in the Lines 2-5 and 7-12. Lines 15-19 specify the subtree structure (for the meaning of `n_lastLeftParIdx` and `n_isLeft` see the node properties f and g in Section 3), mainly by stating that `n1` must be the root of a subtree and `n2` must be a descendant of `n1` and a right-most leaf node (a solution or contradiction). The value of `n_leafs` is the number of leafs (solutions and fails) visited so far during the search process. Property `n_solutions` gives the number of solutions seen so far in the search process. The filter criterion of a high solution density is defined in Line 20, where we use a density threshold of 50% (Line 13). Line 21 initiates the solution process, i.e. the filter evaluation.

[1] If there are no nodes satisfying a filter model, the search tree is shown by VCS just as before and without additional decorations.

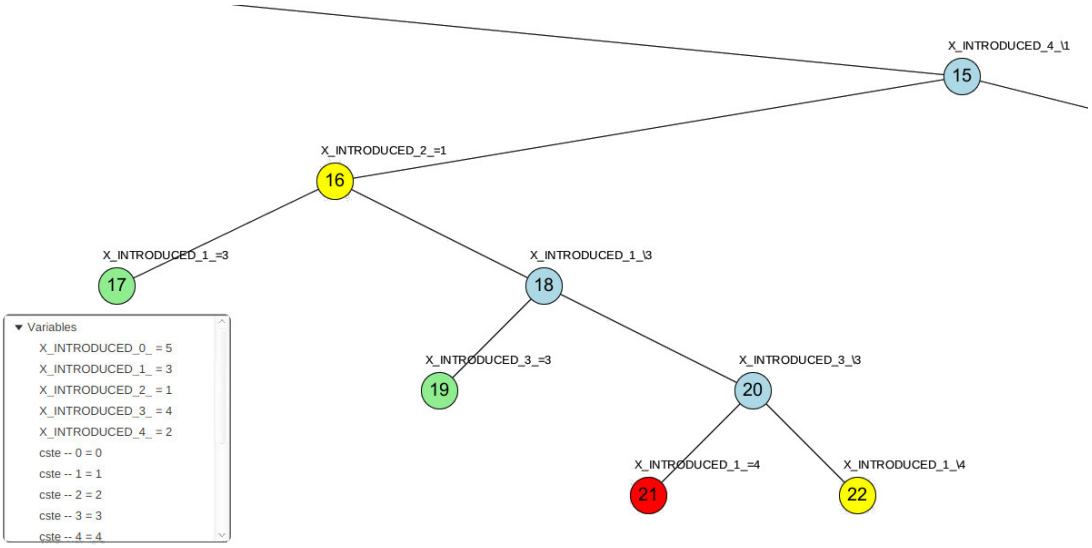

Figure 3: Filtering a search tree for subtrees with a solution density of at least 50%.

```
1  var int: n1_id;
2  var int: n1_lastLeftParIdx;
3  var int: n1_isLeft;
4  var int: n1_leafs;
5  var int: n1_solutions;
6  var int: n2_id;
7  var int: n2_lastLeftParIdx;
8  var int: n2_isLeft;
9  var int: n2_leafs;
10 var int: n2_solutions;
11 var int: n2_isSolution;
12 var int: n2_isContradiction;
13 int: pctg = 50;
14 % minimum solution percentage
15 constraint n1_id < n2_id;
16 constraint n2_isLeft = 0;
17 constraint (int_eq(n1_id,
       n2_lastLeftParIdx) \/
18 (n1_isLeft = 0 /\ int_eq(
       n2_lastLeftParIdx,
       n1_lastLeftParIdx)));
19 constraint ((n2_isSolution = 1) \/
        (n2_isContradiction = 1));
20 constraint (n2_solutions -
       n1_solutions) * 100 div (
       n2_leafs - n1_leafs) >= pctg;
21 solve satisfy;
```

Listing 3: Filtering subtrees with high solution density.

Listing 4 is a filter for finding nodes which perform a strong domain reduction. First, in Lines 1 and 3, the tree nodes `n1` and `n2` are declared. The property `n_X_INTRODUCED_2_dom` provides the domain size of a variable `X_INTRODUCED_2` when node `n` is reached. The index of the parent node of a node is specified by

```
1  var int: n1_id;
2  var int: n1_X_INTRODUCED_2_dom;
3  var int: n2_id;
4  var int: n2_parentId;
5  var int: n2_X_INTRODUCED_2_dom;
6  int: pctg = 30;
7  constraint int_eq(n2_parentId,
       n1_id);
8  constraint 100 - (
       n2_X_INTRODUCED_2_dom * 100 div
       n1_X_INTRODUCED_2_dom) >= pctg;
9  solve satisfy;
```

Listing 4: Filtering subtrees with a strong domain reduction.

`n_parentId`. The constraint in Line 7 ensures that `n1` is the direct parent node of node `n2`. In Line 8 the criterion of domain reduction is determined according to Definition 5 and compared against a certain threshold value (here 30%, Line 6).

The extension of VCS by our new node filtering mechanism allows to debug and analyze search heuristics in detail. When the user wants to compare two search trees (and thus get insights about differences of search heuristics), he can define problem oriented criteria based on node properties, solve his problem CSP P and filter the search trees by the filter criteria defined in a filter CSP F. VCS supports a visualization of the filtered node sets and subtrees and provides further data on the tree nodes including the node properties.

6 CONCLUSION

In this paper we introduced a new method for the assessment and comparison of search heuristics at the evaluation of FD-CSPs. For this, we define filter model CSPs, which describe criteria to filter subtrees of search trees with specific properties. The comparison and systematic analysis of CSP search trees can help to understand the preconditions for good and successful heuristics for specific applications and in general and may help to improve dynamic search heuristics. At this, our main observation was that such filters for search trees are themselves just again CSPs. Thus, we realized a method to define filter CSPs and to handle these by a constraint solver which makes our method very flexible.

Our new method was integrated into the Visual Constraint Solver tool, which allows to present CSPs and the solution process interactively, step-by-step through a suitable visualization. With the extension by filter models, VCS now became a tool for debugging CSPs, comparing search strategies, and understanding search in detail.

Future Work. Currently, VCS is in a prototypical implementation state; in the future we will further improve and optimize this tool, its workflow and expand its application area. Considering the implementation of the filtering feature, improvement is needed for handling large CSPs with high numbers of variables and constraints together with (more complex) filters. This combination yields a growth of constraints and data and can quickly become a memory bottleneck. We need to investigate ways of handling this problem, e.g. by more appropriate data structures, internal constraint representations, and early node set pruning.

Another direction of future work is to provide the user with descriptions of tree patterns for an easier specification of filter CSPs (e.g. for the a pattern for subtrees like Lines 15-19 in Listing 3).

Furthermore, currently only the predefined search heuristics of the Choco solver are taken into consideration as search heuristics. An extension of our approach to other search strategies like domain splitting is desirable.

REFERENCES

Bels, M. (2022). Evaluation und Visualisierung von Entscheidungsbäumen und Variablen- und Wertauswahlheuristiken im Visual Constraint Solver. Bachelor Thesis, BTU Cottbus-Senftenberg.

Buckenauer, D. (2019). Redesign des Tools Visual Constraint Solver (VCS). Bachelor Thesis, BTU Cottbus-Senftenberg.

Choco (2022). Choco - an open-source java library for constraint programming. https://choco-solver.org/. last visited 2022-10-14.

Dechter, R. (2003). *Constraint Processing*. Elsevier Morgan Kaufmann.

Li, X. and Epstein, S. L. (2010). Visualization for structured constraint satisfaction problems. In *Visual Representations and Reasoning, Papers from the 2010 AAAI Workshop, Atlanta, Georgia, USA, July 11, 2010*. AAAI.

Mak, A. (2022). Constraint graph visualization. http://users.cecs.anu.edu.au/~anthonym/cgv.pdf. last visited 2022-10-14.

Marriott, K. and Stuckey, P. (1998). *Programming with Constraints. An Introduction*. The MIT Press.

MiniZinc (2022). The constraint modeling language minizinc. https://www.minizinc.org/. last visited 2022-10-14.

Reda, A. (2020). Visualisierung von Arrays in Constraint-Satisfaction-Problemen. Master Thesis, BTU Cottbus-Senftenberg.

Simonis, H., Davern, P., Feldman, J., Mehta, D., Quesada, L., and Carlsson, M. (2010). A generic visualization platform for CP. In Cohen, D., editor, *Principles and Practice of Constraint Programming - CP 2010*, volume 6308 of *Lecture Notes in Computer Science*, pages 460–474. Springer.

Sinz, C. and Dieringer, E. (2005). Dpvis - A tool to visualize the structure of SAT instances. In Bacchus, F. and Walsh, T., editors, *Theory and Applications of Satisfiability Testing - SAT 2005*, volume 3569 of *Lecture Notes in Computer Science*, pages 257–268. Springer.

Stuckey, P. J., Marriott, K., and Tack, G. (2020). *MiniZinc Handbook*. https://www.minizinc.org/doc-2.6.3/en/index.html, last visited 2022-10-14.

van Beek, P. (2006). Backtracking search algorithms. In Rossi, F., van Beek, P., and Walsh, T., editors, *Handbook of Constraint Programming*, chapter 4, pages 85–134. Elsevier.