Constraints (2016) 21:77–94
DOI 10.1007/s10601-015-9202-1

Visual search tree profiling

Maxim Shishmarev[1] · Christopher Mears[1] ·
Guido Tack[1,2] · Maria Garcia de la Banda[1,2]

Published online: 16 August 2015

Abstract Understanding how the search space is explored for a given constraint problem – and how it changes for different models, solvers or search strategies – is crucial for efficient solving. Yet programmers often have to rely on the crude aggregate measures of the search that are provided by solvers, or on visualisation tools that can show the search tree, but do not offer sophisticated ways to navigate and analyse it, particularly for large trees. We present an architecture for *profiling* a constraint programming search that is based on a lightweight instrumentation of the solver. The architecture combines a visualisation of the search tree with various tools for convenient navigation and analysis of the search. These include identifying repeated subtrees, high-level abstraction and navigation of the tree, and the comparison of two search trees. The resulting system is akin to a traditional program profiler, which helps the user to focus on the parts of the execution where an improvement to their program would have the greatest effect.

Keywords Constraint programming · Search tree · Profiling · Comparison · Visualisation

✉ Maxim Shishmarev
maxim.shishmarev@monash.edu

Christopher Mears
chris.mears@monash.edu

Guido Tack
guido.tack@monash.edu

Maria Garcia de la Banda
maria.garciadelabanda@monash.edu

[1] Faculty of IT, Monash University, Melbourne, Australia

[2] National ICT Australia (NICTA) Victoria, Melbourne, Australia

1 Introduction

Modern approaches to solving combinatorial problems focus on developing a *model* that describes the problem in terms of parameters, variables, constraints and (optionally) an objective function. The parameters can later be instantiated with input data describing a particular *instance* of the problem. To solve an instance, the programmer selects the constraint *solver* that will be used to handle the constraints in the model and the *search strategy* used to explore its search space. We refer to the combination of model, input data, solver and search strategy as the *constraint program*. Importantly, the same problem can be solved using many different constraint programs. Efficiency, measured in terms of how quickly a (good enough) solution to the problem can be found, depends crucially on the combination of model, solver, and search strategy selected by the programmer.

Finding the best combination of model, solver, and search is a very challenging, iterative process, particularly for real-world problems with large-scale input data. This process is a classic instance of *profiling*, where the workflow typically involves three steps that are iterated: (1) we *observe* the behaviour of the program; (2) we develop a *hypothesis* about why certain unwanted or unsatisfactory behaviour occurs; (3) we *modify* the program to test the hypothesis (by observing a *change in behaviour*).

Performance profiling tools are designed to support and speed up this process in a variety of ways. For constraint programs the existing profiling tools (e.g. [1, 3, 8, 11, 14, 18, 21, 22]) mostly focus on visualising the search tree (for those based on tree-search), visualising constraints and variables and their domains, showing solutions or gathering crude aggregate measures. While these tools can provide very useful information, they are also somewhat limited. In particular, most (with the exception of CPViz [22]) require close coupling between the profiling tool and the solver or search strategy, thus making it difficult for the profiling tool to integrate new solvers and search strategies, or to compare different solvers. Also, the information provided by many of these profiling tools leaves the user to explore the search tree in a relatively unguided way, which is particularly problematic for large trees. Finally, and most importantly, none of these tools support the *comparison* of different executions when solving the same problem, but with one or more aspects of the model, solver, or search strategy modified – with the exception of recent work which compares the behaviour of propagators across executions [26].

This paper presents an architecture and tool set specifically designed to support programmers during the performance profiling workflow. In particular, the presented profiling tool enables programmers to *extract* the information they need to build hypotheses; and it helps them *validate* their hypotheses by identifying and visualising the effect of changes in the program (be it changes in the model, solver, or search strategy). The former is achieved by providing different views and analyses of the execution as well as efficient navigation tools to help users focus their attention towards the parts of the execution that might be worth modifying. The latter is achieved by providing two tools, one that can *replay* searches using a different model or solver, so that the user can isolate the effect of a change; and a second tool that can visually "merge" the common parts of two executions and allows users to explore those parts that differ. The idea of replaying search was developed simultaneously with that of Van Cauwelaert et al. [26], who use a similar method to compare propagators.

The architecture is designed to easily accommodate new solvers and search strategies. It is fully implemented, andavailable under an open-source license at https://github.com/cp-profiler.

The rest of the paper is organised as follows. Section 2 provides a brief summary of the required background. Section 3 introduces the proposed architecture. Section 4 illustrates how this architectures helps programmers extract information from a single execution. Section 5 illustrates how it helps programmers form hypotheses and validate them by supporting the comparison of two executions. Section 6 provides a brief summary of the implementation. Section 7 discusses related work. Finally, Section 8 presents our conclusions.

2 Background

A constraint satisfaction problem consists of a set of variables, each with a domain of possible values, and a set of constraints that restrict the combinations of values the variables can take. The task is to assign values to all variables such that all constraints are satisfied. A constraint optimisation problem adds to this an objective function whose value is to be minimised or maximised.

Constraint programming systems often solve constraint problems using a combination of propagation and tree search. The propagation engine uses the constraints and the current domains of the variables to infer smaller domains. If it detects a constraint that cannot be met, or a variable with an empty domain, it reports failure. If all variables have a single value and all constraints are satisfied, it reports success. Otherwise, the system begins to search.

A typical tree search process splits the problem into two or more subproblems that partition the search space. Each subproblem is the same as its “parent” but with an additional constraint, which we refer to as the *branching decision*. The propagation engine is run on each subproblem, and further splitting is done if necessary. In this way, the search moves towards subproblems that can be immediately detected as being satisfiable or unsatisfiable. This process implicitly defines a *search tree* rooted by the original problem where each node represents a branching decision, which is used as the node’s label.

Different branching decisions and different actions when reaching failure result in different kinds of searches. For example, in a simple depth-first search the tree search continues branching until either a solution or a failure is found/reached. In the case of a failure, the search backtracks up the tree, undoing previous decisions until it reaches a node where an alternative decision can be made. In a Restart Based Search, the tree search is interrupted occasionally (e.g., after reaching a certain node limit or encountering a failure), at which point the search goes back all the way up to the root node to explore a different area of the search tree. This way the search does not get stuck in an area of the tree with no solution or little information to learn, which is particularly important for large problems. Some solvers also offer a parallel search mode, with several threads running concurrently, each exploring a particular part of the tree. This does not only serve as an effective exploration technique, but also as a way to support modern multi-core CPUs. A variation of Parallel Search is a Distributed Search, where the work is performed across several computers.

In addition, some modern solvers (e.g. CPX [9], Chuffed [5] or MinisatID [15]) combine tree search with clause learning technology (e.g. Lazy Clause Generation [9]). This involves recording the reasons for each propagation step and using these upon encountering a failure to derive additional constraints called no-goods, which encode the reasons for the failure. No-goods allow the search to *backjump* through the search tree (i.e., go back to the node in the search tree that caused the failure), and also to improve propagation later in the search – in both cases possibly pruning large parts of the search tree.

3 Profiling architecture

This section presents a high-level overview of the architecture of our constraint profiling tool chain. Details regarding its components and how they are used to support the profiling workflow are given in the subsequent sections.

As shown in Fig. 1, the architecture is centred around the *broker*, which receives data regarding the search state from the executing solver. In turn, the broker can either send the data directly to different visualisation/analysis tools or write the data into a database. Similarly, when any of the visualisation/analysis tools requests data, the broker can directly ask the solver or retrieve it from the database. In this way, visualisation/analysis tools are decoupled from the data source and are, thus, independent of whether the execution is proceeding in real time or has already been completed. Storing the execution data also means that the analysis can be continued or repeated at a later point in time without re-running the solver. The decoupling of the solver from the broker is realised using a simple network protocol, which also enables the broker to receive data from several sources simultaneously. This feature is important for profiling parallel exploration techniques, including distributed search across multiple computers.

In addition to the pure structure of the search tree, partial information regarding the state of the solver at each search node can also be recorded. Examples for such additional information can be the variable domains, the amount of domain reduction caused by constraint propagation, or the actual solution in the case of solution leaf nodes. This information can be used to analyse the current execution, and to test hypotheses in a different execution about potential changes in model, solver or search strategy (see Section 5 for an example where search decisions are used to replay the search of a different model). Sending this additional information is optional in order to keep the communication overhead minimal when the information is not needed.

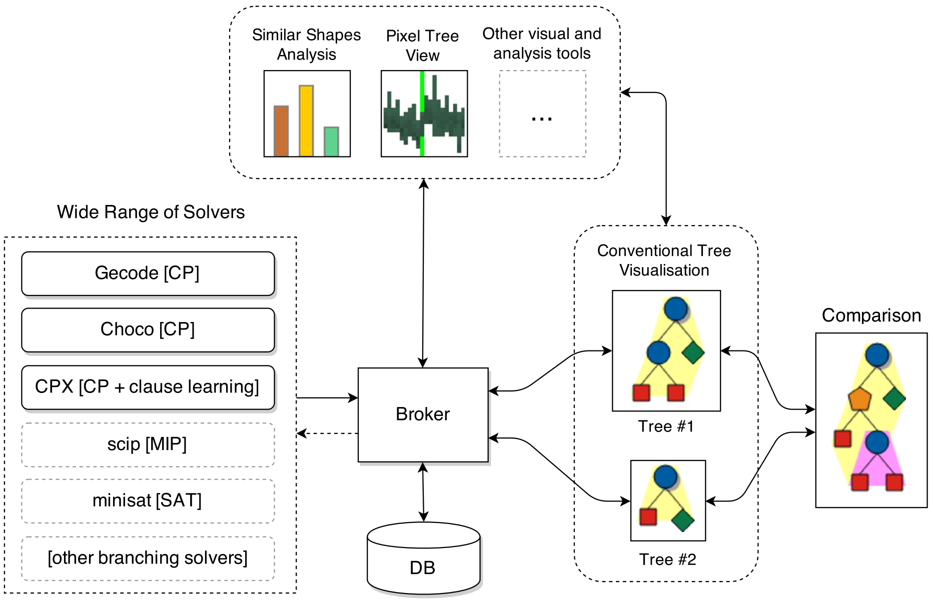


Fig. 1 Architecture of the profiling toolchain

The current implementation of our system includes several visualisation and analysis components. Some of these allow users to find subtrees that are repeated within a single search (see Section 4.1), some give users a high-level overview of how the search behaves over time (see Section 4.2), and others allow users to compare two executions (see Section 5). Note that we have modified the conventional search tree visualisation (i.e. where solver states are represented by nodes, and edges show parent-child relation) to support different types of search (see Section 4.3), such as restart-based search, parallel search and backjumping.

We have so far implemented interfaces to the broker for three different solvers: the Gecode CP solver [19], the CPX clause learning solver [9], and the Choco CP solver [13]. See Section 6 for more on this and other implementation details.

4 Profiling a single execution

A visual representation of the search tree is one of the standard tools that is used to extract information from the solving process that can be used for profiling constraint programs. Whenever the execution of a program takes more time than expected, one would like to analyse the search tree to determine which parts of the search possibly cause the undesired behaviour and, hopefully, which changes to the program might speed up its execution.

Our implementation of this form of visualisation is based on the tool *Gist* integrated in Gecode [20]. Similar to Gist, our visualisation (see e.g. the top half of Fig. 2) shows: nodes where a solution or failure has been reached as diamonds and squares, respectively; nodes with children as circles, indicating there are still some decisions to be made; and entire failed subtrees as triangles.

A major disadvantage of this kind of search tree visualisation as an effective profiling tool is that while the tree may *contain* the information we seek, it is often so large that we struggle to *find* the interesting parts of the tree. This section explores how we can focus on interesting parts and navigate large trees, and how the tree visualisation has been adapted to handle the trees produced by learning solvers, parallel searches and restart-based searches.

4.1 Improving focus

One of the ways to facilitate the analysis of a program's execution is to provide users with some guidance regarding the parts of the search tree on which to focus their attention. This can be achieved in a variety of ways, each useful for detecting different problems. For example, CPViz [22] provides an invariant checker that can be used to detect nodes where the constraint propagation performed by the solver is not as strong as it could be and lead to extra search that might be costly. That way, the user can focus on those search nodes that show poor propagation.

Our focusing approach aims at identifying entire subtrees of the search that show interesting behaviour, by detecting similar subtrees that occur repeatedly in the search tree. This can, for example, indicate the presence of symmetries and the need for some symmetry breaking strategy. To achieve this, our profiler identifies repeated subtrees that have the same *shape*. The shape of a subtree describes how far the subtree extends horizontally at each vertical level. That is, for each level of the subtree, the shape records the horizontal position (in pixels, relative to the root of the subtree) of the leftmost and rightmost nodes at that level. The reason for choosing this particular definition of shape is that it is the one that is already used by the layout algorithm for drawing the tree, and therefore can be obtained

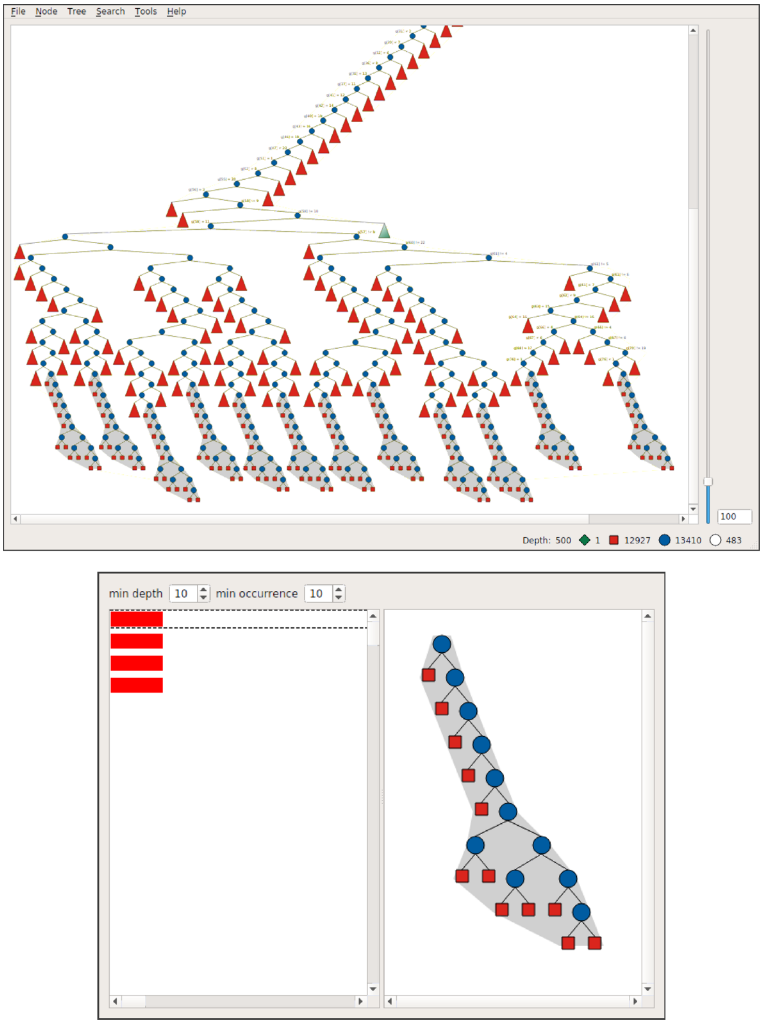

File Node Tree Search Tools Help
100
Depth: 500 1 12927 13410 483
min depth 10 min occurrence 10

Fig. 2 Viewing similar subtrees

without any overhead. Note that this definition disregards the exact decisions taken by the solver in each branch, and captures the outside contour of the tree, without distinguishing between the inner nodes. As a result, this definition allows efficient computation and comparison of the shapes and permits subtrees with similar, but not exactly the same, structure to be matched. The tool then sorts the similar subtrees by their size and number of occurrences, enabling the user to quickly find subtrees of interest.

We illustrate this with an example on the Social Golfers problem, using the model `golfers1` from the MiniZinc benchmarks library.[1] In this problem, a cohort of golfers are to be partitioned into smaller groups over several rounds such that no two golfers play

[1] See https://github.com/MiniZinc/minizinc-benchmarks

together more than once. Figure 2 shows the result of identifying a large set of repeated subtrees in the search for the 4-5-5 instance of the problem. The window on the bottom shows the results of the shape analysis. The profiler allows users to set (top left corner) the minimum size of the subtree of interest and the minimum number of occurrences. For any choice of size and occurrences, the profiler shows a set of horizontal bars, each corresponding to a particular subtree of those characteristics, with the length of the bar indicating the number of occurrences. If the user clicks in one of these bars, the associated search tree is shown (in the figure, the top bar is selected, which is the default). The window on the top shows the part of the search tree where the occurrences appear (highlighted in grey background).

Based on this information we can now formulate a hypothesis. By viewing the branching decisions that lead to these subtrees, we can see that they are the same except for the transposition of some of the groups within a round. By the definition of the problem, the groups of golfers within a round are symmetric and, therefore, transposing them has no effect; consequently, there is unnecessary repeated search of the same subtree. Having identified the presence and cause of this redundant search, we can eliminate it by adding a symmetry breaking constraint on the rounds – the original `golfers1` model has constraints that break the symmetry *within* each group in each round, but not between *different* groups of the same round. Specifically, we add a constraint that forces the groups within each round to be ordered; that is, the "smallest" golfer in the first group is less than the smallest golfer in the second group, and so on for all adjacent groups. The extended model can be found as `golfers1b` in the MiniZinc benchmark suite.

Finally, after adding this constraint to break the symmetry we would like to verify our hypothesis and confirm that the repeated subtrees have been eliminated. One way is to run the similar subtree analysis again and check that the trees no longer appear. Another approach in our profiler is to use *path bookmarks*. In the search tree for the original model we can record the sequence of search decisions that lead to a point of interest – in this case, one of the symmetric repeated subtrees. Then, we can instruct the profiler to follow this same path in the search tree of the improved model to find the same subtree, or to confirm that it has been eliminated. In general, this feature allows users to save interesting points in the tree and retrieve them later, and also allows them to find corresponding points in related but different search trees.

4.2 Different ways to view and navigate the search

Another way to facilitate the analysis of large search trees is to provide users with different ways in which to view and navigate the search tree.

The size of search tree visualisations is usually managed by automatically collapsing any failing subtree (that is, sub-trees without any solutions). This is indeed the case for our profiler, where collapsed subtrees are shown as triangles. In addition, we allow users to collapse subtrees based on a user-defined number of nodes N instead, resulting in a view that reflects the amount of exploration effort. Any subtree with fewer than N nodes is collapsed, and the size of the resulting triangle is scaled according to the number of nodes in the subtree. This way the size of a subtree can be estimated by the number of triangles, while the entire tree can still be shown in a compact way. Figure 3 shows a search tree that has been collapsed with N set to 1000. In this view, it is easy to identify the root node of the subtree where most of the work has been spent (as highlighted in the figure). When collapsing subtrees without a size limit, the highlighted subtree would be shown as a single triangle since it does not contain any solution, and therefore would be indistinguishable from

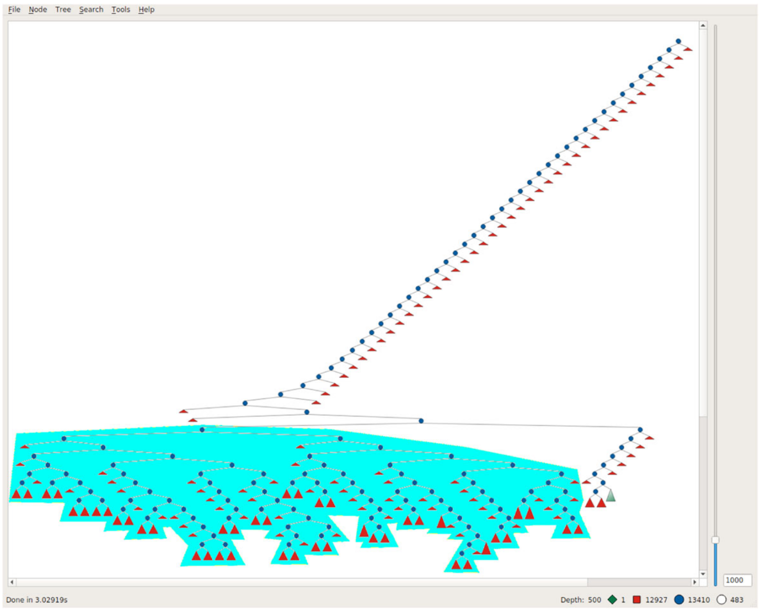

File Node Tree Search Tools Help
1000
Done in 3.02919s
Depth: 500 1 12927 13410 483

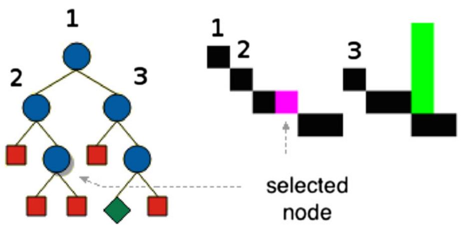
Fig. 3 Collapsed subtrees by size ($N = 1000$)

all other collapsed subtrees. Collapsing subtrees by size thus makes it possible to determine at a glance where most of the search effort was spent.

Indented Pixel Tree Plots [2] provide another way of viewing a search tree, where the behaviour of the search over time is more prominent. Pixel trees are designed to clearly represent hierarchical structures and substructures, their sizes and their depths, and to scale up to millions of elements. They represent leaf nodes as single square objects (e.g. pixels), inner nodes as vertical lines, and edges only implicitly by the indentation between the objects (lines and squares): Parent nodes are placed immediately to the left of their subtree and the leaves of each subtree are grouped to the rightmost position. As demonstrated in [2], the resulting plot can be easily scaled vertically and horizontally, can incorporate additional information using different colours, and it is easy to interact with (expand, collapse, filter, etc). Further, it allows users to perform tasks such as detecting similar subtrees and estimating which of two subtrees is larger, even for very large trees.

As an example, consider the basic search tree shown in Fig. 4. For each node from the search tree on the left, there is a corresponding pixel (square) on the pixel-tree on the right. (For illustration, the pixels are scaled up; also some nodes/pixels are annotated with labels, so that nodes/pixels with the same label represent the same entity). Pixel trees preserve the

Fig. 4 Basic search tree (*left*) and corresponding pixel tree (*right*)

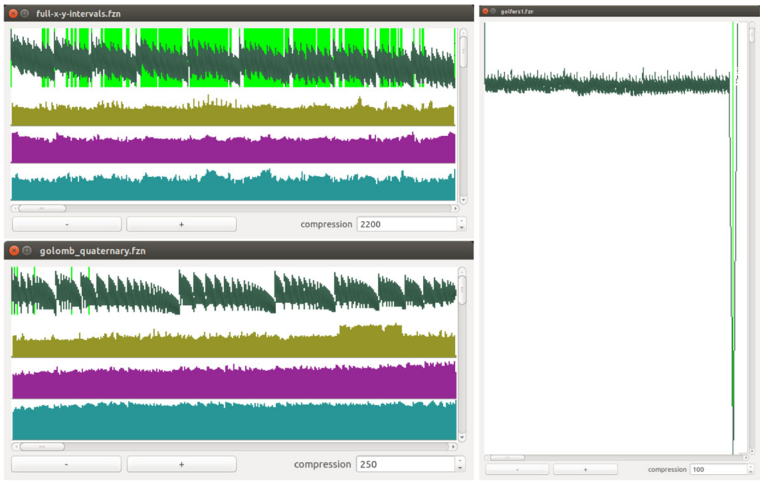


vertical position of a node, or depth: e.g. node ‘2’ lies on the second level in the search tree, and so does the corresponding node on the pixel tree. However, horizontal position of nodes on the pixel-tree represent depth-first-search exploration order: node ‘2’ is positioned to the right of its parent ‘1’, which is not the case in the conventional tree visualisation on the left. Nevertheless, the pixel tree preserves the hierarchical structure: e.g. the immediate ancestor of node ‘3’ can be found as the closest node to the left of node ‘3’ that lies one level above (node ‘1’ in this case).

While not being as easy to interpret as conventional trees (although studies show [2] that the ability to read such trees improves greatly with just a bit of practice), two of their properties are particularly important for us: a) they show very clearly the relationship between node exploration and time (i.e. a node is guaranteed to have been explored prior to all the nodes to its right); b) they scale well horizontally, which is crucial for showing large trees.

The conventional tree visualisation sacrifices the node-time relationship in favour of showing a node’s ancestry more clearly. Note also that conventional trees, when presented in a compressed way (usually by focusing on one part of the tree and hiding/collapsing the rest) do not give a good overview of the whole tree. A pixel tree, on the other hand, can be compressed in a different way: adjacent nodes can be squashed together to form vertical lines of pixels. If some of these nodes also happen to be on the same depth level, they will be collapsed into a single pixel, and the colour of the pixel will represent the node density in that case. Of course, a pixel tree that is compressed this way does not convey the same amount of information as the original, but still preserves the overall structure.

We highlight the location of each solution in the pixel tree with a vertical line. This allows for a clear representation of the amount of effort between solutions, and can be used to immediately identify different patterns of behaviour. Consider for example the three pixel trees depicted in Fig. 5. The rightmost one corresponds to finding the first solution in `golfers1`, the top-left one corresponds to the $Full_{XY}$ model of the All Interval Series problem provided in [4] for $n = 12$, and the bottom-left one corresponds to the quaternary version of the Golomb ruler model from [25]. Clearly, they depict very different search behaviours, from one that follows a steep, single assignment path except for a significant



compression 2200
compression 250
compression 100

Fig. 5 Different search behaviours shown with pixel trees

amount of search performed mid-way through the tree (golfers1), to one whose fractal-like pattern (quaternary Golomb ruler) we believe might be associated with a generate-and-test-style search due to lack of propagation.

The modularity of the architecture facilitates using the profiler as a test bed for experimental measurements. For example, the two left windows in Fig. 5 show, under each of the pixel trees, three histograms of extra information associated with the node (or nodes when compressed, which is always the case except for toy instances) depicted in the pixel tree: time taken to produce the nodes, average domain size at those nodes, and average amount of domain reduction from the parent to the nodes. While the interpretation of these measures is not yet clear, we have found some interesting patterns. For example, the tree in Fig. 6 has a distinct area rich in solutions, and the corresponding time-per-node histogram underneath shows that nodes in that area were significantly harder to compute (propagation takes longer to reach a fixpoint), while in the rest of the search the node rate appears to be practically constant.

The pixel tree can also be used to drive navigation of the conventional tree view: clicking on a point in the pixel tree will highlight and scroll to the corresponding part of the conventional tree view. This permits the user to identify an interesting region via the high-level pixel tree view, and then immediately examine it in detail in the conventional tree view.

4.3 Visualising different kinds of search

The profiling architecture is flexible enough to support search methods that are not straightforward depth-first traversals of the search tree. We have adapted the conventional tree visualisation to display backjumps, parallel search and restart-based search.

To display backjumps it is necessary to distinguish a node that was not explored from one which was deliberately skipped due to a backjump in a learning solver. We mark such skipped nodes with lighter grey squares (see Fig. 7), which allows users to see how much of the search space has been skipped thanks to the inferred no-good.

The extension for parallel solvers is straightforward, as the broker receives data using a network protocol and does not require the data to come from a single source. The profiler can therefore readily consume data from solvers working in parallel. Figure 8 shows the search tree for a Golomb ruler with 7 marks solved using Gecode's parallel search engine. Each solver process is assigned its own colour and the subtrees are highlighted accordingly. Note that the rightmost triangle has a white background despite being contained in the coloured subtree on the right. This indicates that the first (white) process did not have any work left and was able to steal some from the fourth process. This kind of visualisation can be useful to see whether specific techniques, developed for distributing work in real time during the search, are effective and result in the expected granularity.

A restart-based search, in effect, explores a new search tree every time it restarts. We display this by considering each new search tree to be the child of a dummy root node,

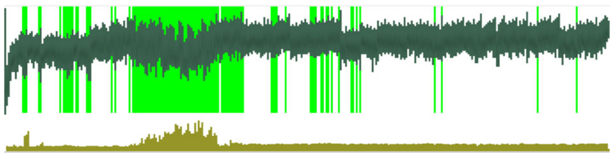


Fig. 6 Histogram showing slower node rate in a solution-rich area

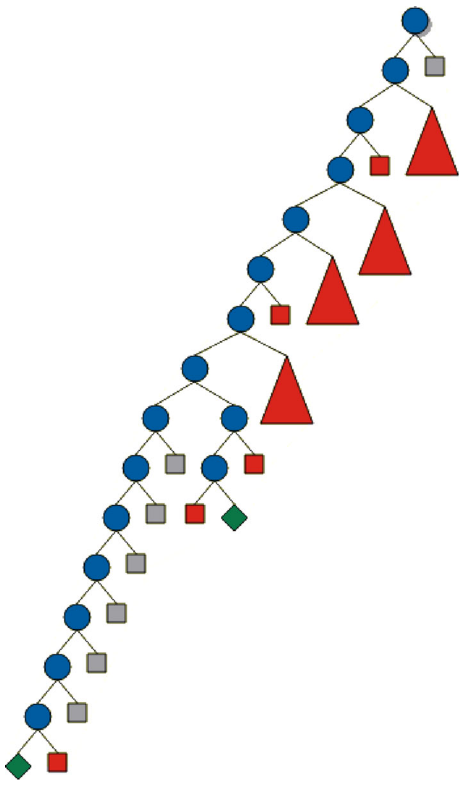


Fig. 7 Solving the Golomb ruler problem (5 marks) with opturion CPX

combining the separate search trees into one super-tree (Fig. 9). Note that white circles represent nodes that are unexplored because search restarted before the solver reached them.

After collapsing the subtrees according to their size (see Section 4.2), the tree shows a high-level view of how the search effort is distributed over the restarts and where the solutions are found. In this case, one can see that most of the work is done during the second-last restart.

5 Comparing several executions

As described earlier, profiling usually involves hypothesising about the causes for undesired behaviour, and then changing the model, search strategy or solver to validate the hypothesis. In order to determine the effect of a given modification and thus select those changes that have the biggest impact, users need to be able to compare the two executions. Currently, this is only possible by means of relatively crude aggregate measures, such as the run-time or the size of the search tree. Our profiling system allows users to more closely examine the differences between two executions. Let us illustrate this by using a small case study as demonstration.

Choi et al. [4] consider several models for the All Interval Series Problem, including the M_Y model, which has variables for the positions of the numbers (referred to as the Y variables), and the $Full_Y$ dual model, which also includes variables to represent the inverse viewpoint (referred to as the X variables) and channeling constraints to connect the two

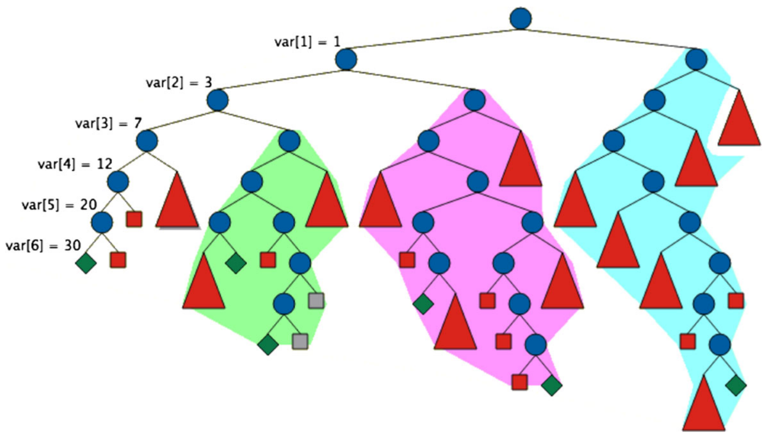


Fig. 8 Execution of Golomb ruler problem (7 marks) with four parallel threads

viewpoints. The idea is that by including both viewpoints, constraint propagation will be stronger. Both models are executed using a first-fail search strategy on the Y variables, that is, branching on the Y variables and choosing at each step the Y variable with the smallest domain.[2] Note that this is a dynamic search strategy, i.e., the individual decisions depend on how much the constraint propagation engine has been able to narrow down the variable domains at each node.

The experimental results of Choi et al. show that the $Full_Y$ model performs significantly better than the M_Y model, both in terms of search space reduction and time reduction. However, from these measures alone one cannot tell what causes the improvement: stronger propagation obtained from the extra constraints, or a difference in the search decisions due to the differences in propagation. We would like to understand where the improvement comes from and, thus, determine whether further improvements are possible.

Our profiler allows us to *replay* the search decisions made in one execution onto a separate execution, that is, execute a possibly different program (where the search variables have not changed) branching on exactly the same decisions as a previously executed program. Note that this is different from simply using the same search strategy in both executions: since the variable domains may be different using different models, the same strategy may make different decisions. In this case, we can take the search decisions made when solving the M_Y model and replay them using the $Full_Y$ model. With the two searches made identical, any differences in execution can be attributed to the propagation strength alone.

Using Gecode, we experimentally compared three variations: the M_Y model using first-fail search, the $Full_Y$ model using first-fail search, and the $Full_Y$ model using a replay of the search recorded for the M_Y model. The results are shown in Table 1.

As found by Choi et al. [4], executing model $Full_Y$ using first-fail is better than executing model M_Y with first-fail. Further, our results show that $Full_Y$ using the replayed M_Y search also performs better than M_Y. Since every decision taken during the execution of $Full_Y$ in the replay search is the same as for M_Y, this shows that the considerable reduction in search space is entirely due to propagation. Interestingly, the replay execution of $Full_Y$ gives better

[2]The $Full_{XY}$ model mentioned above, is the same as $Full_Y$ but executed searching first on the X and then on the Y variables.

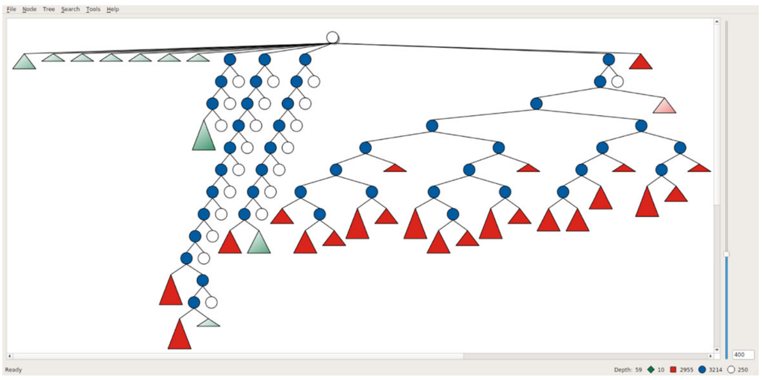


Fig. 9 Restart-based search with subtrees collapsed

results than those obtained using first-fail. This behaviour is unexpected – why should the decisions that were guided by the *weaker* propagation yield *better* behaviour than those guided by *stronger* propagation?

We can use the search tree comparison feature of our profiler, shown in Fig. 10, to pinpoint where the two executions diverge. Our comparison algorithm analyses two trees by traversing both of them in lockstep until they diverge. This point of divergence is shown in the tree as a pentagon, and the diverging subtrees are shown as children of that pentagon node. The traversal then backtracks to find the next point of divergence, until the trees are exhausted. In the figure the box on the bottom displays a list of all points where the trees diverge; each column corresponds to one execution and, for each column, the row shows the sizes of the two differing subtrees. In the highlighted case there is a large difference between the two subtrees. This prompted us to select that case for examination by clicking on that row. Upon selection, the two subtrees are highlighted in different colours, centred in the screen, and each tree is annotated with its first search decision. Another point later in the search (not shown) has a similar difference of approximately 1000 nodes. In both cases, the $Full_Y$ search has chosen to branch on variable Y_3 and the replayed search has chosen Y_{11}. The ability to focus on where the search strategy has the most impact can suggest to the modeller where some insight into the best strategy may be gained.

6 Implementation

As with other profiling tools for constraint solvers, our architecture requires solvers to be modified to produce the necessary information. Importantly, the changes we require are minimal and not onerous, as we only need solvers to generate small amounts of easily

Table 1 All interval series problem, number of failures to find all solutions

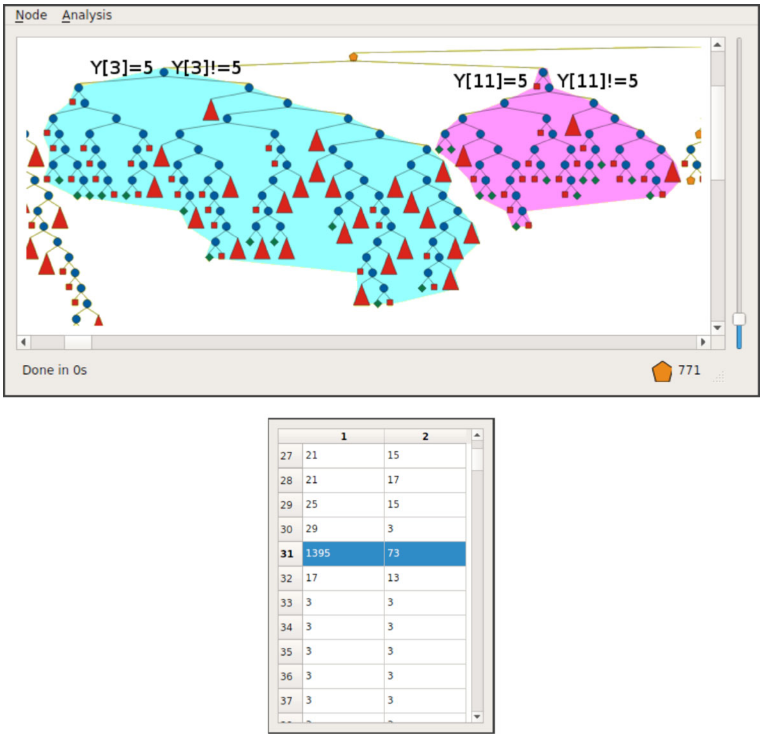
n	M_Y first-fail	$Full_Y$ first-fail	$Full_Y$ replay
11	254472	3954	3504
12	1483831	13341	11528
13	9183752	49243	41654

formatted data every time the search branches. For instance, adding the profiling capabilities to Gecode and CPX (the two have a similar architecture) required only about ten lines of code for each of the search engines. The integration of Choco (a solver whose source code we were unfamiliar with) for most non-advanced profiling capabilities required adding about 20 lines of code took about 5 hours to integrate (with most of the time spent on familiarising with the source code).

A typical unit of information sent from a solver to the profiler to describe each node in the search must contain the following fields: the node's identifier, the identifier of its parent, the number of the node's children, the identifier of the node among its siblings, and its status. This is sufficient to build the search tree structure. However, solvers are encouraged to provide additional information for each node: branching decisions (labels), domain size, etc.

Each message is packed in a binary format using the Protocol Buffers serialisation library [10], which can construct messages of dynamic size and content (there is no cost for sending empty information if the solver does not provide certain fields). Additionally, solvers can add arbitrary information to be attached to a search tree visualisation.

It is important to make sure the message encoding is efficient, especially in the case of distributed search, where the bandwidth of the network can be a limiting factor. Average



Done in 0s

771

	1	2
27	21	15
28	21	17
29	25	15
30	29	3
31	1395	73
32	17	13
33	3	3
34	3	3
35	3	3
36	3	3
37	3	3

Fig. 10 Search tree comparison, showing where two search trees diverge

size of a message with a short label is about 32 bytes, but if the node sent is, for example, a solution node, one can attach text describing the solution to the message and the size of these particular messages will increase.

The communication between solvers and broker is implemented using the cross-platform socket library ZeroMQ [12], which is fast and supports bindings for most modern programming languages. The latter means our architecture is practically independent of the implementation language of the solver and, while Gecode, CPX and the profiling tools are all written in C++, the solvers written in other programming languages (like Choco, which is written in Java) can be integrated as easily. For solvers in Java and C++ we provide an API for easy integration, encapsulating all network-related routines. Using socket-based communication also allows for the solver and the profiler to run on different computers, so that the profiler need not contend for resources with the solver.

For the sake of simplicity, we use a SQLite database for storing and retrieving executions. It is 'server-less' (in contrast to most other SQL-based engines) and lightweight, but still provides reasonable performance.

The visualisation of a search tree in our architecture will inevitably introduce overheads resulting from the need to communicate information between the solver and the visualiser, store the tree in memory, and draw the tree. How much these extra operations will affect the execution time depends on how much time the solver spends in processing the nodes. The overhead for communication is constant per node; therefore, the smaller the solving time per node, the bigger the communication overhead. To measure the overhead in a worst-case scenario, we have built a simple model where there is no propagation and, therefore, the solver spends virtually no time in processing each node. This provides an upper bound to the overhead introduced by our architecture. Table 2 shows the execution times (wall-clock time in seconds) needed to solve different instances of this model with no visualisation, visualising using Gist (which only works with the Gecode solver), and visualising using our tool. As expected, both visualisations introduce significant overheads for this case, with our profiler being approximately 8 times slower than the original execution. The improved performance of our profiler compared to Gist is due to the overhead being spread over several components which can run in parallel – the communication and visualisation run in separate threads from the solver. Nonetheless, the results also show that we can have solver-independent visualisation of multi-million node search trees at a reasonable cost.

Of course, on more realistic models most of the time is spent on propagation as can be seen in Table 3, which shows the results for two instances of the well-known Golomb Ruler problem with our visualisation taking approximately 40 % longer than the original execution. In this case our profiler is slightly slower than Gist, since the parallelisable part is a much smaller proportion of the overall runtime. Importantly, our current implementation

Table 2 Time taken in a worst-case scenario (no propagation)

Size (nodes)	No visual.	Gecode Gist (solver-specific visual.)	Our tool (solver-independent visual.)
2.1M	0.5s	10.5s	3.1s
4.2M	1.0s	21.9s	9.3s
8.4M	2.0s	43.6s	15.3s

Table 3 Time taken for the Golomb ruler (GR) problem

Instance (size)	No visual.	Gecode Gist (solver-specific visual.)	Our tool (solver-independent visual.)
GR 11 marks (640k)	5.8s	7.1s	7.6s
GR 12 marks (5.4M)	56.0s	69.9s	71.9s

is a prototype; there remains the opportunity to further improve performance and reduce the overhead.

The full implementation is available under an open-source license at https://github.com/cp-profiler.

7 Related work

Our work builds on the results of previous research on profiling (and debugging) of constraint programs to improve runtime performance (e.g., [1, 3, 6–8, 11, 14, 18, 21, 22]). While the focus of this paper is tree search visualisation, it is important to note that there has been significant work in related areas such as matrix-based visualisations of the variables and/or their domains (e.g. [3, 6, 14, 22, 27]), visualisation of the constraint-network (e.g. [16, 17, 24]), and problem-specific and/or custom visualisations (e.g. [8]). All these kinds of visualisations are also important for gaining insight into the program's execution, particularly when combined with tree search visualisation. We have started to integrate some of these approaches (variable/domain evolution and constraint networks) into our profiler.

Regarding tree search visualisation, the closest work is that of CPViz [22], a generic visualisation platform implemented in Java that can operate with any solver that outputs an appropriate XML execution trace. CPViz is a very complete system that integrates not only different views of the search tree (conventional, compact and TreeMap), but also variable/domain visualisations, custom visualisations, and an invariant check visualisation. The main differences with respect to our profiler are as follows. First, CPViz is a post-mortem system, a decision made to ensure solver-independence (and inherited from [7]). While we also want to be able to integrate solvers easily, we believe it is crucial to allow certain, restricted forms of interaction that can provide deep insight without significantly increasing solver integration (e.g., those that allow us to replay an execution). Second, we support user navigation through the tree by connecting it to the pixel tree, which makes it much easier to select individual solutions and get an overview of the tree in relation to solving time. Third, we can focus the attention of the user towards interesting parts of the tree, such as those displaying similar tree shapes. Most importantly, our profiling architecture supports the comparison of trees, which allows users to determine the effect of changes to a given program. As far as we know, no other tool (including CPViz) can support this.

As mentioned before, our conventional tree-visualiser is an extension of Gist [20]. While Gist supports real-time profiling as well as the capability to control the search process step-by-step, this requires it to be deeply coupled with the solver (Gecode). The same is true for its ancestor Oz Explorer [18] and existing tools for SAT solvers [23, 24]. Again, none of these tools supports comparison of different search executions, or facilitates navigation.

8 Conclusion

We have presented an architecture and system for the visual profiling of tree search. The presented tools allow users to extract information about solver performance by helping them navigate the search tree and focus on interesting parts of the tree, as well as to validate their hypotheses thanks to the possibility to replay search and compare two different executions.

The architecture allows for solvers to be easily integrated without imposing an unreasonable overhead on the program's execution. The modularity of its design supports the combination of many different visualisation and analysis components. This opens up several avenues for future work.

The no-goods produced by learning solvers are crucial to good performance, but it is difficult to explore the set of learned no-goods. We intend to develop analysis and visualisation tools for such no-goods that may yield new insights into difficult problems; for example, to associate the no-goods learnt at a specific failure in the search tree. We also seek to incorporate mixed integer programming and SAT solvers into the system, to shed more light on how these typically black-box solvers operate. Indeed, the architecture supports the comparison of executions of different kinds of solver (e.g. constraint programming and SAT) on the same problem. Further, we plan to add more visualisation and analysis components for use with all kinds of solvers, particularly focus-based components guided by the statistical analysis of the search tree.

Acknowledgments We thank the anonymous reviewers who helped us by pointing out which ideas were not clearly described as well as providing other useful comments. NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council. This research was partly sponsored by the Australian Research Council grant DP140100058.

References

1. Bauer, A., Botea, V., Brown, M., Gray, M., Harabor, D., & Slaney, J. (2010). An integrated modelling, debugging, and visualisation environment for G12. In *CP 2010. LNCS*, (Vol. 6308 pp. 522–536): Springer.
2. Burch, M., Raschke, M., & Weiskopf, D. (2010). Indented pixel tree plots. In *Advances in Visual Computing* (pp. 338–349): Springer.
3. Carro, M., & Hermenegildo, M. Tools for constraint visualisation: The VIFID/TRIFID tool. In: Deransart et al. [6], pp. 253–272.
4. Choi, C.W., Lee, J.H.M., & Stuckey, P.J. (2007). Removing propagation redundant constraints in redundant modeling. *ACM Transactions on Computational Logic*, *8*(4). doi:10.1145/1276920.1276925.
5. Chu, G.G. (2011). *Improving combinatorial optimization. Ph.D. thesis*: University of Melbourne.
6. Deransart, P., Hermenegildo, M.V., & Maluszynski, J. (Eds.) (2000). *Analysis and visualization tools for constraint programming, constraint debugging (DiSCiPl project), LNCS*, Vol. 1870: Springer.
7. Deransart, P. (2004). Main results of the OADymPPaC project. In B. Demoen, & V. Lifschitz (Eds.), *ICLP, LNCS*, (Vol. 3132 pp. 456–457): Springer.
8. Dooms, G., Van Hentenryck, P., & Michel, L. (2007). Model-driven visualizations of constraint-based local search. In *CP 2007, LNCS*, (Vol. 4741 pp. 271–285): Springer.
9. Feydy, T., & Stuckey, P.J. (2009). Lazy clause generation reengineered. In I.P. Gent (Ed.), *CP, Lecture Notes in Computer Science*, (Vol. 5732 pp. 352–366): Springer.
10. Google (2015). Protocol buffers. https://developers.google.com/protocol-buffers/.
11. Goualard, F., & Benhamou, F. Debugging constraint programs by store inspection. In: Deransart et al. [6], pp. 273–297.

12. iMatix (2015). ZeroMQ. http://zeromq.org.
13. Jussien, N., Rochart, G., & Lorca, X. (2008). Choco: an open source Java constraint programming library. In *CPAIOR'08 Workshop on Open-Source Software for Integer and Contraint Programming (OSSICP'08)* (pp. 1–10).
14. Meier, M. (1995). Debugging constraint programs. In *CP'95, LNCS*, (Vol. 976 pp. 204–221): Springer.
15. MinisatID team (2015). MinisatID solver. https://dtai.cs.kuleuven.be/software/minisatid.
16. Müller, T. (1999). Practical investigation of constraints with graph views. In K. Sagonas, & P. Tarau (Eds.) *Proceedings of the International Workshop on Implementation of Declarative Languages (IDL'99)*.
17. Newsham, Z., Lindsay, W., Liang, J.H., Czarnecki, K., Fischmeister, S., & Ganesh, V. (2014). SATGraf: Visualizing community structure in boolean SAT instances. https://ece.uwaterloo.ca/ vganesh/EvoGraph/Home.html.
18. Schulte, C. (1997). Oz explorer: a visual constraint programming tool. In L. Naish (Ed.), *ICLP* (pp. 286–300): MIT Press.
19. Schulte, C., Lagerkvist, M.Z., & Tack, G. (2006). Gecode: Generic constraint development environment. http://www.gecode.org.
20. Schulte, C., Tack, G., & Lagerkvist, M.Z. (2015). Modeling and programming with Gecode. http://www.gecode.org/doc-latest/MPG.pdf.
21. Simonis, H., & Aggoun, A. Search-tree visualisation. In: Deransart et al. [6], pp. 191–208.
22. Simonis, H., Davern, P., Feldman, J., Mehta, D., Quesada, L., & Carlsson, M. (2010). A generic visualization platform for CP. In *CP 2010, LNCS*, (Vol. 6308 pp. 460–474): Springer.
23. Sinz, C. (2004). Visualizing the internal structure of SAT instances (preliminary report). In *SAT*.
24. Sinz, C., & Dieringer, E.M. (2005). DPvis – a tool to visualize the structure of SAT instances. In F. Bacchus, & T. Walsh (Eds.), *Theory and applications of satisfiability testing, lecture notes in computer science*, (Vol. 3569 pp. 257–268). Berlin Heidelberg: Springer. doi:10.1007/11499107_19.
25. Smith, B.M., Stergiou, K., & Walsh, T. (1999). Modelling the Golomb ruler problem. Tech. rep., University of Leeds School of Computer Studies.
26. Van Cauwelaert, S., Lombardi, M., & Schaus, P. (2015). Understanding the potential of propagators. In *Integration of AI and OR Techniques in Constraint Programming* (pp. 427–436): Springer.
27. Wallace, M.G., Novello, S., & Schimpf, J. (1997). ECLiPSe : a platform for constraint logic programming. *ICL Systems Journal*, *12*(1).