Visualizations to Explain Seach Behavior

Leave Authors Anonymous
for Submission
City, Country
e-mail address

Leave Authors Anonymous
for Submission
City, Country
e-mail address

Leave Authors Anonymous
for Submission
City, Country
e-mail address

ABSTRACT

In this paper, we propose visualizations that track the progress and behavior of backtrack search when solving an instance of a Constraint Satisfaction Problem. The goal of our visualizations is to provide insight in the difficulty of the particular instance at hand as well as in the effectiveness of various strategies for enforcing consistency during search. To this end, our visualizations track the number of backtracks and the number of calls to a consistency algorithm per depth of the search tree and superimpose the two measures while distinguishing effective and wasteful consistency calls. Using these numbers, we automatically derive qualitative regimes summarizing the evolution of the search process over time. We show that these instruments provide new insights into the performance of search on a particular instance and into the effectiveness of the various strategies for enforcing consistency during search. We present WORMHOLE, an extendable, solver-agnostic visualization tool that we built as a platform to implement these mechanisms. Currently WORMHOLE provides a 'post-mortem' analysis of search, but our ultimate goal is to provide an 'in-vivo' analysis and allow the user to intervene and guide the search process.

ACM Classification Keywords

H.5.2 Information Interfaces and Presentation: User Interfaces; I.2.8 Artificial Intelligence: Problem Solving, Control Methods, and Search

Author Keywords

explainable artificial intelligence; constraint satisfaction problems; search; consistency algorithms; satisfiability

INTRODUCTION

In this paper, we propose a new perspective and visualization tools to understand and analyze the behavior of the backtrack-search procedure for solving Constraint Satisfaction Problems (CSPs). Backtrack search is currently the only sound and complete algorithm for solving CSPs. However, its performance is unpredictable and can differ widely on similar instances. Further, maintaining a given consistency property during search has become a common practice and new strategies for dynamically switching between consistency algorithms are being investigated [2, 3, 4, 8, 12, 13, 15, 19, 23, 24, 33, 35, 36, 41]. While consistency algorithms can dramatically reduce the size of the search space, their impact on the CPU cost of search can vary widely. This cost is currently poorly understood, hard to predict, and difficult to control.

In this paper, we propose three tools and their visualizations as a first step towards graphically summarizing and explaining the performance of search:

1. We track the number of *backtracks per depth* (BpD) at each level of the search process to understand where and how search struggles and where it smoothly proceeds.

2. To understand the impact of enforcing a given consistency property, we track the number of *calls to the consistency algorithm per depth* (CpD) in the search tree. Further, we split these calls into three categories: those that yield domain *wipeout* (i.e., detect inconsistency), those that effectively *filter* domains without detecting a dead-end, and those that yield *no filtering* (i.e., constitute a wasted effort).

3. To summarize the behavior of search over time, we structure the the temporal evolution of the above two measurements into a *history* of qualitatively equivalent *regimes* by using three criteria that characterize growth rate and shape evolution.

We implement the above mechanisms in a visualization system called WORMHOLE that allows users to interactively examine the performance of search and graphically compare the performance of various consistency algorithms on the same instance. WORMHOLE is a first step towards building a library of visualization tools aimed at providing an insight into the strengths and weaknesses of current algorithms for solving CSPs.

While our system does not generate verbal explanations, we claim that the graphical tools that it provides directly 'speak' to a user's intuitions. Our long-term goal is to allow users to actively intervene in the search process itself, trying alternatives and mixing strategies while observing their effects on the effectiveness of problem solving.

This paper is structured as follows. We first review background information and the relevant literature. Then, we describe our two core contributions, namely, the visualizations and the structuring of the evolution of search into regimes of equivalent behaviors. We illustrate the usefulness of each of these

CHI'16, May 07–12, 2016, San Jose, CA, USA

ISBN 123-4567-24-567/08/06…$15.00

DOI: http://dx.doi.org/10.475/123_4

two contributions with a case study. Finally, we describe the architecture of WORMHOLE and conclude this paper.

BACKGROUND

Constraint Satisfaction Problems (CSPs) are used to model many combinatorial decision problems of practical importance in Computer Science, Engineering, and Management. A CSP is defined by a tuple (X, D, C), where X is a set of *variables*, D is the set of the variables' *domains*, and C a set of *constraints* that restrict the combinations of values that the variables can take at the same time. A solution to a CSP is an assignment of values to variables such that all constraints are simultaneously satisfied. Determining whether or not a given CSP has a solution is known to be NP-complete. The *constraint network* of a CSP instance is a graph where the vertices represents the variables in the CSP and the edges represent the constraints and connect the variables in their scope.

Backtrack (BT) search is currently the only sound and complete algorithm for solving CSPs. In order to reduce thrashing, which is the main malady of search, it is common practice to enforce a given *consistency property* after every variable instantiation. This procedure reduces the size of the search space by deleting, from the variables' domains, values that cannot appear in a consistent solution given the current search path (i.e., conditioning). In recent years, the research community has investigated *higher-level consistencies* (HLC) as inference techniques to prune larger portions of the search space at the cost of increased processing effort [6, 11, 14, 26], leading to a trade-off between the search effort and the time for enforcing consistency. We claim that our visualizations are insightful tools for understanding such a trade-off.

Prior research on search visualization has appeared in the Constraint Programming literature. The DiSCiPl project provides extensive visual functionalities to develop, test, and debug constraint logic programs such as displaying variables' states, effect of constraints and global constraints, and event propagation at each node of the search tree [10, 30]. Many useful methodologies from the DiSCiPl project are implemented in CP-Viz [31] and other works [29]. The OZ Explorer displays the search tree allowing the user to access detailed information about the node at each tree node and to collapse and expand failing trees for closer examination [27]. This work is currently incorporated into Gecode's Gist [28]. The above approaches focus on exploring the search tree (as well as a problem's components) while our work proposes particular projections (i.e., views, summaries) of the data *reflecting* (i.e., compiling) the cost and the effectiveness of both search and enforcing consistency. We believe that these visualizations are orthogonal and complementary.

Tracking the search effort by depth of the search tree was first proposed by Epstein et al. [13] for the number of constraint checks and values removed per search level and by Helmut et al. [31] for the number of nodes visited in CP-Viz (also used for solving a packing problem [32]). We claim that the number of constraint checks, values removed, and nodes visited are not accurate measures of the thrashing effort. Indeed, the number of constraint checks varies with the degree of the variables. The number of values removed and nodes visited vary with the size of the domain. In contrast, we claim that the number of *backtracks per search depth* (BpD) provides a more faithful representation of the thrashing effort, which is exactly the aspect of search that we aim to characterize.

Recently, techniques have appeared in Constraint Processing for dynamically choosing between a set of consistency properties based on the CPU time spent on exploring a given subtree [4]. We claim that we better track the effectiveness of such decisions by following the number of backtracks per depth (BpD) and the number of *consistency calls per depth* (CpD) rather than the CPU time of searching a given subtree.

In the SAT community, inprocessing (in the form of the application of the resolution rule) interleaves search and inference steps [17, 42]. Resolution is allocated a fixed percentage of the CPU time (e.g., 10%) and sometimes its effectiveness is monitored for early termination. We believe that inference should be targeted at the 'areas' where search is struggling rather than following a predetermined and fixed effort allocation. We claim that the visualization provided by WORMHOLE can be used to identify where inference is best invested.

VISUALIZATIONS: SEARCH EFFECTIVENESS

In this section, we introduce three visualizations based on BpD and CpD to analyze the performance of search.

Number of Backtracks per Depth (BpD)

The BpD chart reflects various aspects of search effectiveness as we illustrate with an example. We consider `4-insertions-3-3` an instance of a coloring problem.[1] We use a backtrack search to find a first solution to this instance using the dom/wdeg variable ordering heuristic [9]. In one experiment, we enforce at each variable instantiation the most commonly used consistency property GAC [20]. In the second experiment, we enforce a stronger consistency property known as POAC [5]. The definitions of the consistency properties GAC and POAC are beyond the scope of this paper: it suffices to say that an algorithm that enforces GAC is generally quick but does little filtering while a POAC algorithm is typically (very) costly but can prune larger subtrees of the search space. In fact, enforcing POAC during search is so costly that, in practice, we use an adaptive version called APOAC [2]. GAC fails to solve this instance within the two-hours time threshold, while APOAC successfully terminates with a solution.

Figures 1 and 2 display the BpD charts of GAC and APOAC, respectively. Comparing these two BpD charts, we see that GAC thrashes around depth 50 with $\max_{\text{BpD}} = 15{,}863{,}603$ backtrack at depth 52. APOAC, which enforces a strictly stronger consistency throughout search, limits the severity of thrashing to only $\max_{\text{BpD}} = 693{,}829$ backtracks at depth 42. Comparing the magnitude of the peaks and their depth, we realize APOAC is able to detect and prune inconsistencies at the shallower levels of the search space. Thus, the investment in a stronger consistency property payed off: the APOAC solves the problem while GAC fails to terminate.

[1] Benchmark `graphColoring-k-insertions` from www.cril.univ-artois.fr/~lecoutre/benchmarks.html.

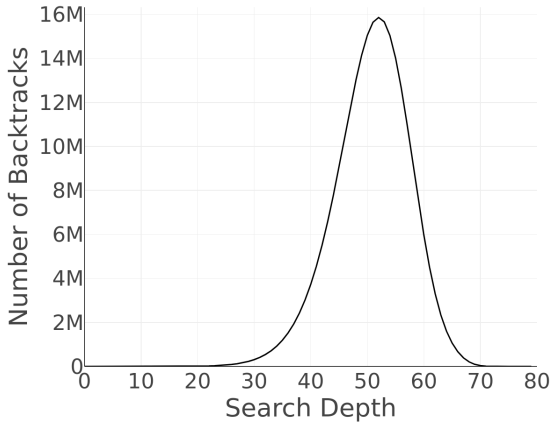

Figure 1. BpD for GAC on `4-insertions-3-3`.

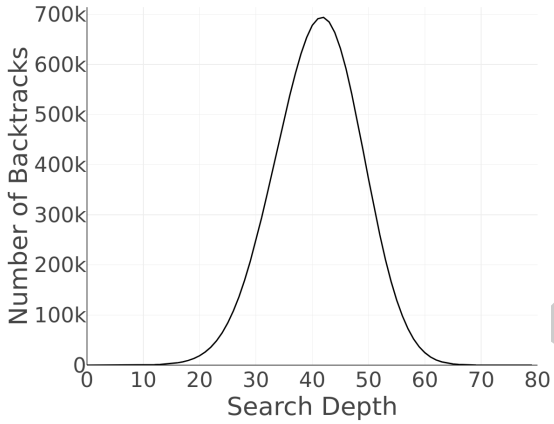

Figure 2. BpD for APOAC on `4-insertions-3-3`.

Table 1 reports the cost of the two search algorithms in terms of their CPU time, the number of nodes visited by search, and the total number of backtracks. The sign '>' indicates that the time threshold was reached before the algorithm could terminate. WORMHOLE displays in a panel these values. Table 1 also reports and the maximum value reached by the respective BpD curves.

	GAC	APOAC
CPU Time (sec)	> 8,099.9	2,981.9
# Nodes Visited	348,276,252	17,078,644
# Backtracks	252,570,526	13,416,093
$\max_{BpD}$	15,863,603	693,829

Table 1. Cost of GAC and APOAC on `4-insertions-3-3`

As we can see from Table 1, it is clear that the 'investment' in a stronger consistency algorithm is worthwhile because APOAC solves the instance in about 50 minutes while GAC does not terminate.

Number of consistency Calls per Depth (CpD)

Naturally, investing in a high-level consistency (HLC) is not always worthwhile. On easy instances, the cost of APOAC can be an overkill. To examine the effectiveness of enforcing an HLC, we propose another visualization, which superimposes, to the BpD chart, the chart reporting the number of calls to POAC per depth (CpD).

Figure 3 shows that the BpD (black) and the CpD (purple) charts of APOAC, as can be expected, largely overlap in shape (modulo their respective ranges shown on both sides of the chart), which is explained by the fact that APOAC is called at every variable instantiation during search.

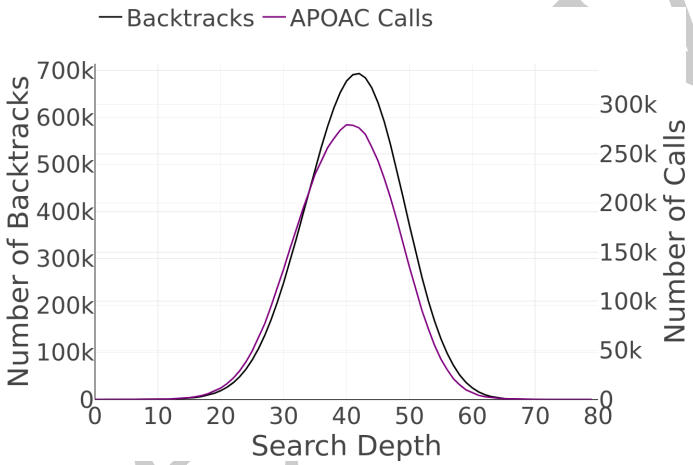

Figure 3. APOAC: Superimposing BpD and CpD on `4-insertions-3-3`

In other hybrid strategies that enforce two or more levels of consistency in the same search, the CpD would allow us to differentiate between the impact of each consistency algorithm, which would be shown in a different color.

Splitting the CpD

We propose a third visualization that more finely splits the CpD into three categories depending on whether calls to POAC resulted in:

1. *a domain wipeout*: indicating that the subtree rooted at the instantiated variable is inconsistent and can be pruned.

2. *filtering but no wipeout*: indicating that the consistency algorithm removed some paths or subtrees rooted at the instantiated variable but other paths and subtrees remain and need to explored before ruling out the current instantiation.

3. *no filtering*: indicating that the consistency algorithm could not remove any path rooted at the current instantiation and, thus, the effort of enforcing the consistency property was entirely wasted.

In Figure 4, these three CpDs are shown in:

1. green: the most effective POAC calls, which prune the subtree

2. blue: POAC calls that prune inconsistent subtrees and reduce the size of the search space, but cannot detect inconsistencies

3. red: POAC calls that result in no filtering and thus are wasteful

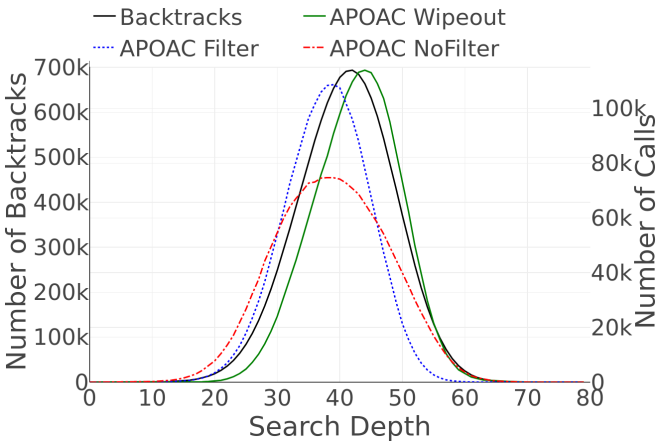

Figure 4. BpD (black) and the CpD split into wipeout (green), filtering (blue), no filtering (red)

In the case of our particular example, we can see that, at deeper levels of search, the wasteful calls (red) to POAC are fewer than than the most effective ones (green). Consequently, not as much time is wasted at deeper levels. This realization explains the ability of APOAC to prevent search from thrashing at deeper search levels and its effectiveness in solving this difficult instance.

REGIMES: SEARCH EVOLUTION

The visualizations discussed above provide interesting information at particular snapshots in time and at the end of search. However, in practice, search can last from a few milliseconds in duration to hours (before timeout). Thus, it is not practical, sometimes impossible, to examine the *evolution* of search from beginning to end. In order to help the user build an understanding of the evolution of search over time, gain insight in the difficulty of the problem at hand, perhaps even detect critical transitions in search behavior, we propose to automatically and dynamically organize the evolution of search over time in terms of a *history of successive episodes that are qualitatively meaningful*,[2] where each episode is a time interval where the behavior remains qualitatively unchanged.

We propose a two-step procedure to build such histories. First, we partition the time duration of the entire search into time intervals based on some criterion of equivalent behavior. We call these time intervals *regimes* [18]. Next, we provide a mechanism to *dilate* the time duration (i.e., by compressing or stretching it) of each regime, then concatenate the dilated regimes into a continuous animation of the history of the search.

DEFINITION 1 (REGIME, HISTORY). *Given a search procedure of total duration T, a given behavior function B and an equivalence relation $\sim$, a regime R_i is a contiguous interval of time $[s_i, e_i] \subseteq T$ during which the search features defining the behavior are qualitatively equivalent by some metric. The search history is a sequence of such regimes.*

$$H = \langle R_1, \ldots, R_n \rangle$$
$$R_i = \langle B_i, [s_i, e_i] \rangle \text{ where } [s_i, e_i] \subseteq T,$$
$$\forall t_i, t_j \in [s_i, e_e] \rightarrow B(t_i) \sim B(t_j)$$
$$T = \bigcup_{i=1}^{n} [s_i, e_i]$$

While the set of desirable and useful behaviors and search features may be limitless, we integrate in WORMHOLE the following three:

1. BASIC: Using the maximum value of BpD, $\max_{\text{BpD}}$, we partition the duration of search into a number of k regimes (k determined by the user), where the largest BpD value exceeds another k^{th} fraction of the value of $\max_{\text{BpD}}$.
2. GROWTH: From the beginning of search, we generate a new regime each time the largest BpD value increases by 10%.
3. SHAPE: From the start of search, we compute the Shannon Entropy of the derivatives over depth of the BpD to represent the relative shape of BpD curve [25]. We start a new regime when this value changes by 20%.

Of the above three regimes, SHAPE can recognize the most interesting changes in the BpD because it recognizes the change of the depth where thrashing occurs. Figures 5 and 6 show two regime progressions of GAC on the `pseudo-aim-200-1-6-4` instance.[3] Figure 5 emphasizes the growth over time of the BpD. Figure 6 highlights three shapes of the BpD within the regime shown at the left of Figure 5.

We generate animations (of a user-specified duration) by applying time dilation on the histories generated with any of the three above behaviors.

Currently, WORMHOLE implements two time-dilation methods, namely, EQUAL and PROPORTIONAL. EQUAL assigns to each regime an equal amount of time, while PROPORTIONAL assigns to each regime an animation time that is proportional to the regime's duration in search. In addition, WORMHOLE allows the user to directly modify the percentage of time that each regime takes independently of the above two dilation methods.

CASE STUDY: EFFECTIVENESS OF INFERENCE

In this section, we discuss a case that demonstrates the usefulness of the visualizations in analyzing search effectiveness. In particular, we use WORMHOLE to understand and compare the behavior of PREPEAK$^+$, a new reactive strategy for enforcing high level consistency during search [39].

To this end, we solve the CSP instance `pseudo-aim-200-1-6-4` studied in Section on "Regimes: Search Evolution" with backtrack search under three settings: (1) maintaining GAC, (2) maintaining POAC, and (3) using PREPEAK$^+$. PREPEAK$^+$ is a hybrid, reactive strategy that mixes calls to GAC and POAC. It is conservative in that

[2] Our use of the term 'history' is in compliance with its initial meaning as proposed by Hayes [16].

[3] Instance `pseudo-aim-200-1-6-4` of the benchmark `pseudo-aim` from www.cril.univ-artois.fr/~lecoutre/benchmarks.html.

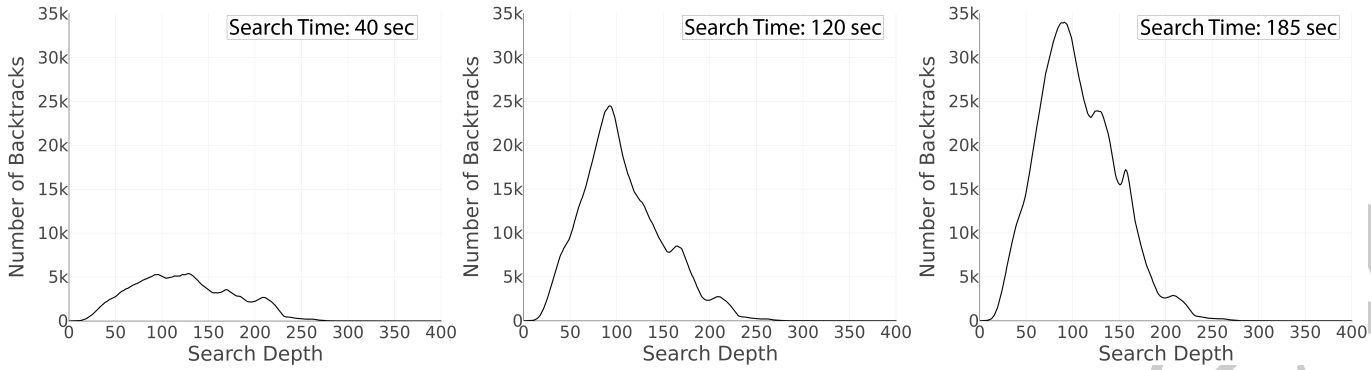

Figure 5. BASIC regimes of GAC on `pseudo-aim-200-1-6-4` (cropped right-edges for space)

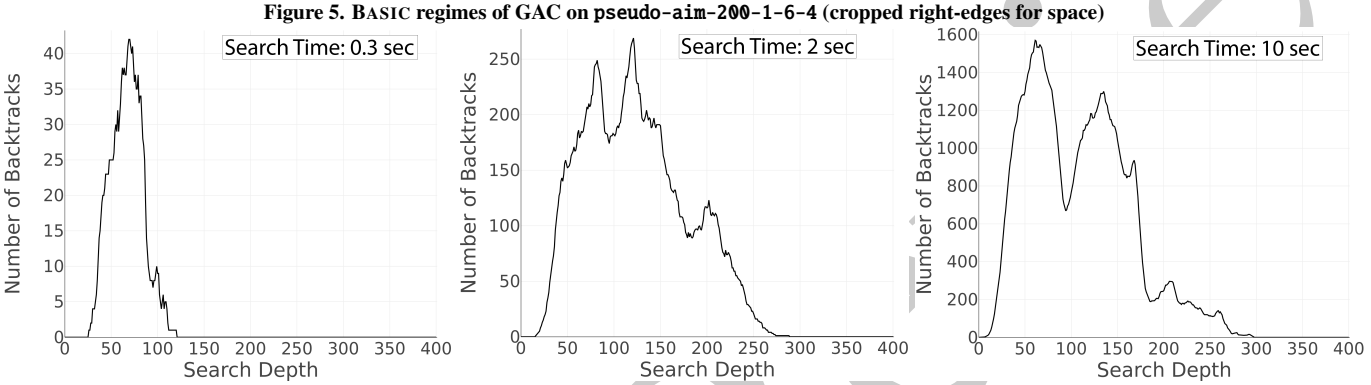

Figure 6. SHAPE regime of GAC on `pseudo-aim-200-1-6-4` (cropped right-edges for space)

it primarily enforces GAC. However, it triggers an HLC, such as POAC, when the number of backtracks per depth (BpD) reaches a given threshold value θ but only when search backtracks to levels shallower than the depth where the threshold is met. PREPEAK$^+$ keeps firing the HLC as long as the BpD at the considered depth is smaller than θ. Furthermore, every time it backtracks, PREPEAK$^+$ updates the values of θ by reducing it or increasing it according to three geometric laws depending on whether the HLC yields wipeout (i.e., it is effective), filters the search space, or yields no filtering (i.e., the HLC calls are wasteful). WORMHOLE reports the costs of three search algorithms as shown in Table 2.

	GAC	APOAC	PREPEAK$^+$
CPU time (sec)	185,045	66,816	17,836
#NV	3,978,074	47,457	284,289
$\max_{\text{BpD}}$	34,023	407	2,421
#HLC calls	NA	7,739	228

Table 2. Cost of GAC, APOAC, PREPEAK$^+$ on `pseudo-aim-200-1-6-4`

Figure 7 shows the BpD for GAC at the end of search. This curve exhibits a peak around depth 100 with $\max_{\text{BpD}} = 34{,}023$ showing that GAC is too weak to filter out bad values: it spends much of its time thrashing around this depth level.

Figure 8 shows the BpD (black) and CpD (colored) curves for APOAC. Examining the BpD curve, we realize that APOAC so effectively prunes the 'bad subtrees' from the search space that it dramatically reduces $\max_{\text{BpD}}$ down to 407 and the location of peak to around depth 75. We see that this instance benefits

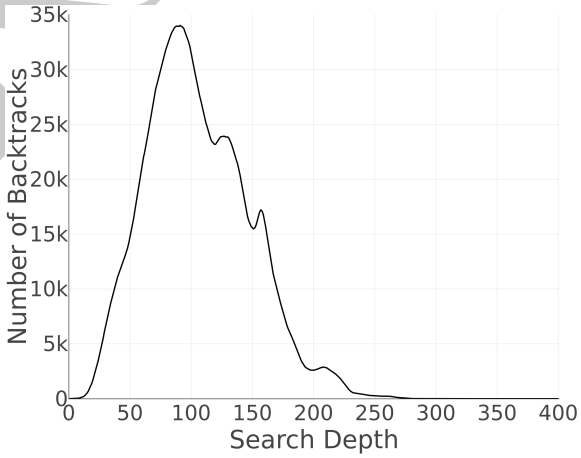

Figure 7. BpD of GAC on `pseudo-aim-200-1-6-4`

from enforcing an HLC such as POAC with a clear benefit on the CPU time (which is reduced by one order of magnitude from GAC). However, by observing the colored curves in Figure 8, we notice that the number of calls to POAC that are ineffective (red curve) are of the same order as those that yield wipeout (green curve). The detailed CpD curves hint to some savings that could be further obtained could one cancel the wasteful calls to POAC.

Figure 9 shows the BpD (black) and CpD (colored) curves for PREPEAK$^+$. PREPEAK$^+$ is conservative in that it calls an HLC only when search thrashes, justifying the cost of a

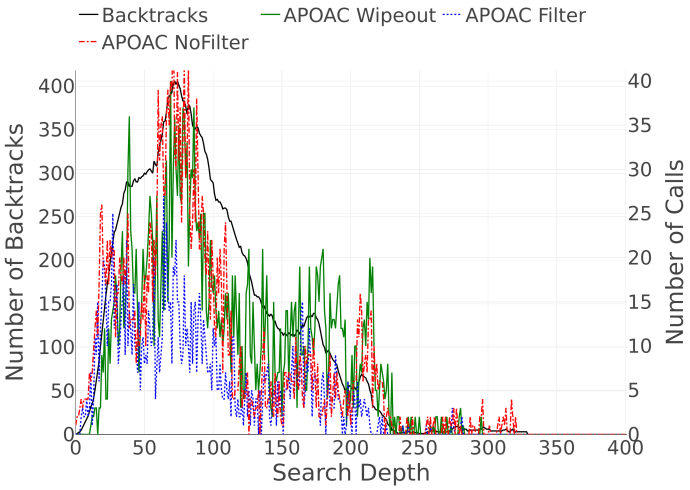

Figure 8. BpD (black) and CpDs (colored) of APOAC on `pseudo-aim-200-1-6-4`

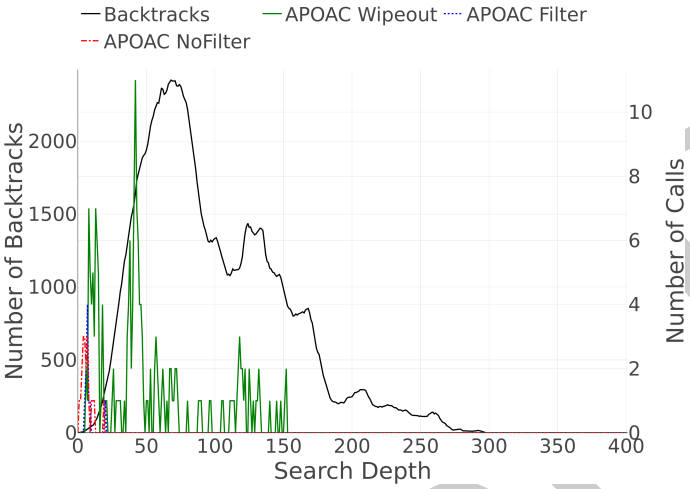

Figure 9. BpD (black) and CpDs (colored) of PREPEAK$^+$ on `pseudo-aim-200-1-6-4`

stronger but more costly consistency algorithm. Indeed, we observe that the peak value of BpD is smaller than for GAC but greater than for APOAC (2,421 versus 34,023 and 407, respectively). However, examining the detailed CpD curves shows that advantage of PREPEAK$^+$: Indeed, the wasteful calls to POAC (red) are almost eliminated and the total calls to POAC are reduced down to 228 for PREPEAK$^+$ from 7,739 for APOAC. This economy in the calls to POAC is immediately translated by the reduction of the CPU time (see Table 2). Thus, despite the fact that PREPEAK$^+$ explores a larger search tree than APOAC (see number of nodes visited) because it does not call the HLC at each variable instantiation, it effectively reacts to thrashing, calling the HLC only when it is needed, but spontaneously reverting to GAC otherwise.

This example illustrates the pertinence of the tools provided by WORMHOLE in visually explaining the behavior of search and the benefits of PREPEAK$^+$.

CASE STUDY: REGIME VISUALIZATION

In this section, we show how the identification and visualization of the regimes that summarize the evolution of search provide insight into the structure of a problem instance and guide the choice of the appropriate type of consistency for solving it.

We consider the CSP instance `mug100-25-3` with 100 variables of a graph-coloring benchmark.[4] We try to solve this instance with the most popular search technique, namely, by enforcing the consistency property GAC at each variable instantiation and using the dom/wdeg for dynamic variable ordering. Search fails to terminate within two hours. The inspection of the BpD chart (Figure 10) reveals the presence of a 'dramatic' peak of the number of backtracks at depth 83.

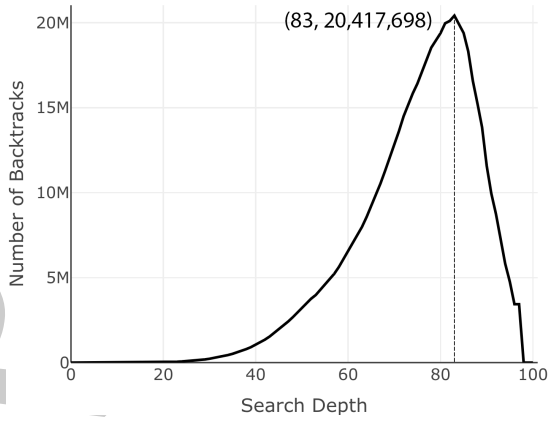

Figure 10. BpD of GAC on `mug100-25-3` after two hours

This behavior hints that search may have made a bad decision at a shallow level of search from which it was never able to recover. In order to gain more insight into the situation at hand, we choose to inspect the various qualitatively distinct regimes that summarize the search 'history.' The setting SHAPE for automatic regime identification uncovers three revealing regimes illustrated with the snapshots at the following time instants: 1 second, 3 seconds, and 10 seconds shown in Figure 11.

The first regime shown in Figure 11 (left) reveals a peak of magnitude 1,595 backtracks at depth 35, which corresponds to instantiating about one third of the variables in the problem. The regime displayed in the middle shows that search has overcome the initial bottleneck in the instance occurring at depth 32 with a magnitude of 4,556 backtracks but is struggling again with a second bottleneck at depth 86 with a magnitude of 1,854 backtracks. The third regime displayed in Figure 11 (right) shows that the severity of the first bottleneck is dwarfed by a dramatic increase in magnitude of the second bottleneck, which reaches 25,637 backtracks at depth 86. As time progresses, the shape of the BpD quickly conforms to the one shown in Figure 10, thus confirming that search never really recovers from the bad decisions done at shallower levels: indeed, search disproportionality invests more efforts in deeper levels (around depth 86) than in shallower ones.

[4]Benchmark `mug` from www.cril.univ-artois.fr/~lecoutre/benchmarks.html

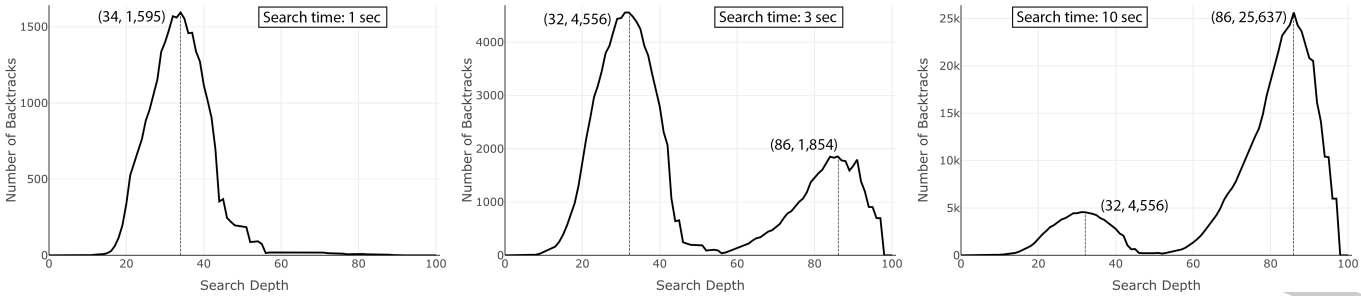

Figure 11. Three regimes of the BpD, qualitatively characterizing the evolution of GAC on instance `mug100-25-3`

Given the time scale of these regimes with respect to the two-hour duration of the experiment (which is typical), it is extremely unlikely that a human user could autonomously have spotted these critical transitions happening so early in the search (at a time scale of seconds). The identification and observation of the two distinct bottlenecks and their evolution entice us to examine the constraint network of this specific instance shown in Figure 12.

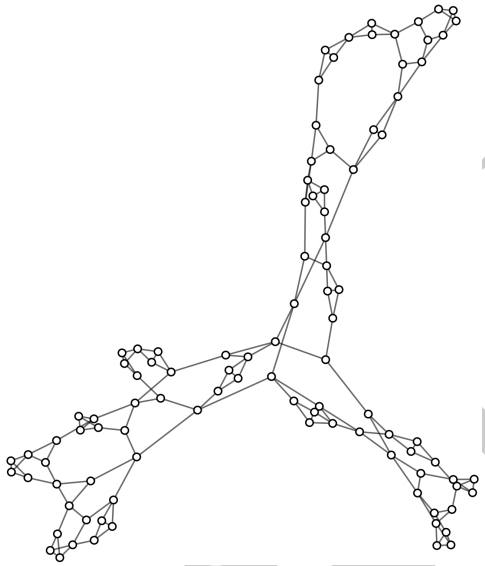

Figure 12. Constraint network of `mug100-25-3`

The examination of this graph unveils the existence of large cycles and many cycles of size three. Previous work has argued that the presence of cycles can be detrimental to the effectiveness of constraint propagation and has shown how triangulation of the constraint network allows us to remedy the situation [36, 37, 40]. With this insight, we choose to enforce the consistency property known as Partial Path Consistency (PPC), during search, instead of GAC because the algorithm for PPC operates on existing triangles and triangulated cycles of the constraint network [7]. Because PPC is too expensive to enforce during search, Woodward proposed a computationally competitive algorithm to enforce a relaxed version of Directional Partial Path Consistency (DPPC$^+$) [38]. By enforcing a strong consistency along cycles, search is now able to detect the insolvability of `mug100-25-3` and terminates in less than 17 seconds.

Figure 13 shows the number of backtracks per depth of search on `mug100-25-3` while enforcing DPPC$^+$. This chart shows a peak of 13,536 backtracks at depth 34.

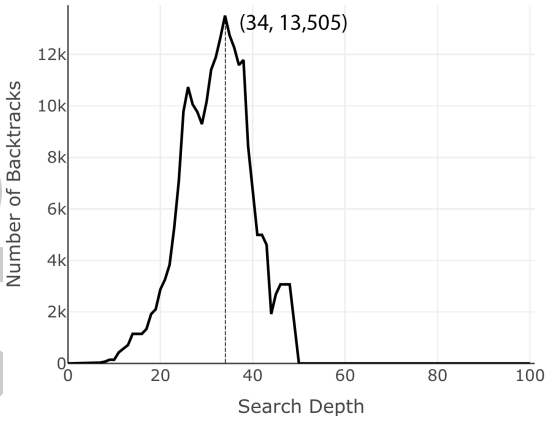

Figure 13. BpD of DPPC$^+$ on `mug100-25-3`

Table 3, which is also displayed by WORMHOLE, allows us to compare the cost of search with GAC and with DPPC$^+$.

	GAC	**DPPC$^+$**
CPU time (sec)	$>$ 8,099.9	16.9
#Nodes visited	683,428,413	288,976
# Backtracks	559,624,248	236,356
max_{BpD}	20,417,698	13,536
max_{BpD} depth	86	34

Table 3. Cost of GAC and DPPC$^+$ on `mug100-25-3`

Above, we showed how the visualizations and regimes generated by WORMHOLE of the search behavior allows us to form a hypothesis about the issues encountered by search and attempt an alternative technique that is usually avoided because of its cost, culminating in successfully solving a difficult benchmark instance. Further, we have circumscribed one source of insolvability to a subproblem of the original CSP, which calls for new visualizations for analyzing and elucidating the bottleneck. The entire experience illustrates how visualizations empower the user to effectively intervene in problem solving.

Without this functionality of WORMHOLE, the exact cause of massive, deep thrashing in this instance would have been difficult to identify and characterize. In solving this instance, WORMHOLE saves us precious time and effort and presents a meaningful story to a human user.

ARCHITECTURE

In this section, we review the architecture of WORMHOLE and explore various optimizations necessary to visualize the investigated features of search through our benchmark problems, due to the magnitude of the data collected.

Figure 14 presents the architecture of WORMHOLE through its various components and their interactions. A generic constraint solver (1) exports data to a JSON-based log file (2) that records the differences in tracked features over time. A user loads a log file into WORMHOLE which parses the data (3) and splits it into various feature-dependent data structures. The Data Store (4) caches these structures and provides access to them during animation generation. In addition, the Data Store sends the final BpD and CpD curves to be displayed in the Per-Depth Chart (5). The user may create new animations, select specific time information, control animation playback, or edit animation parameters through an animation control-panel (6). Upon editing the parameters of an animation the animation is regenerated from the data store (7).

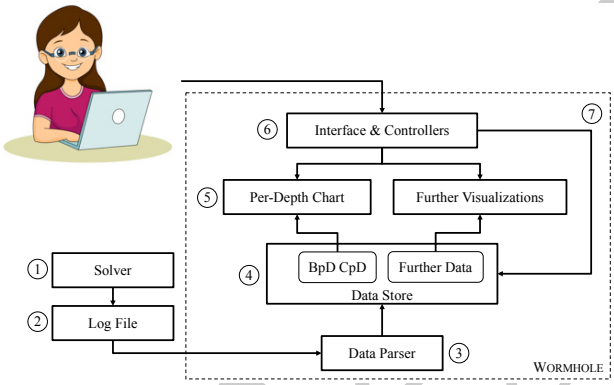

Figure 14. Architecture of WORMHOLE

We have designed WORMHOLE to be portable, using the web technologies HTML, JavaScript, and CSS based on the React-Redux model [1, 34]. In this framework, React manages view rendering and user interaction while Redux governs the application state. While the standard Redux paradigm works adequately well for log files with a limited magnitude of data (10's of MB), a majority of our benchmarks require far larger log files (100's of MB), which create an intolerably slow user experience on such systems. In addition, JavaScript's single-threaded, event-loop paradigm may prevent user interaction during periods of heavy computation, a phenomenon known as blocking.

To address these performance bottlenecks, we enhance the system as follows. In WORMHOLE we mitigate blocking by incorporating web workers [21], a modern feature of the web browser; web workers offload a majority of CPU-intensive processing to a separate process. In this process, which we refer to as the data process, we perform initial parsing, animation pre-fetching, and specific-time frame loading, which all demand analyzing whole feature data sets.

To accelerate frame loading during animation, we pre-fetch all animation data in the data process and transfer it using JavaScript Transferables: data structures for rapidly transferring large data sets between web workers and the main thread [22]. Further, when parsing the log file, we perform regular sampling on the BpD and CpD data in order to reduce the work needed to calculate specific-time frames. When users request BpD and CpD data for a specific time, we perform nearest neighbor interpolation on these samples before incorporating supplemental difference data.

By leveraging our optimization techniques, in practice our system enables users to interactively explore the evolution of BpD and CpD curves in a web browser.

CONCLUSION

WORMHOLE offers a number of new visualization and animation techniques that allow the user to explore and understand the behavior of backtrack search and compare the performance of different algorithms. We are currently applying our techniques to Choco, a popular constraint solver that uses binary search (2-way branching) and will expand this effort to other public-domain constraint solvers to allow users to compare the performance of such solvers.

ACKNOWLEDGMENTS

Intentionally left blank.

REFERENCES

1. Dan Abramov and Andrew Clark. 2017. Redux. `https://redux.js.org/`. (2017). Accessed: 2017-04-21.
2. Amine Balafrej, Christian Bessiere, El-Houssine Bouyakhf, and Gilles Trombettoni. 2014. Adaptive Singleton-Based Consistencies. In *Proceedings of AAAI-2014*. 2601–2607.
3. Amine Balafrej, Christian Bessiere, Remi Coletta, and El-Houssine Bouyakhf. 2013. Adaptive Parameterized Consistency. In *Proceedings of* 19 *th International Conference on Principle and Practice of Constraint Programming (CP'13) (LNCS)*, Vol. 8124. Springer, 143–158.
4. Amine Balafrej, Christian Bessière, and Anastasia Paparrizou. 2015. Multi-Armed Bandits for Adaptive Constraint Propagation. In *Proceedings of the 24 th International Joint Conference on Artificial Intelligence*. 290–296.
5. Hachemi Bennaceur and Mohamed-Salah Affane. 2001. Partition-k-AC: An Efficient Filtering Technique Combining Domain Partition and Arc Consistency. In *Proceedings of* 7 *th International Conference on Principle and Practice of Constraint Programming (CP'01) (LNCS)*, Vol. 2239. Springer, 560–564.
6. Christian Bessière, Kostas Stergiou, and Toby Walsh. 2008. Domain Filtering Consistencies for Non-Binary Constraints. *Artificial Intelligence* 172 (2008), 800–822.

7. Christian Bliek and Djamilla Sam-Haroud. 1999. Path Consistency for Triangulated Constraint Graphs. In *Proceedings of the 16 th International Joint Conference on Artificial Intelligence*. 456–461.
8. James E. Borrett, Edward P.K. Tsang, and Natasha R. Walsh. 1996. Adaptive Constraint Satisfaction: The Quickest First Principle. In *Proceedings of the 12 th European Conference on Artificial Intelligence*. 160–164.
9. Frédéric Boussemart, Fred Hemery, Christophe Lecoutre, and Lakhdar Sais. 2004. Boosting Systematic Search by Weighting Constraints. In *Proceedings of the 16 th European Conference on Artificial Intelligence*. 146–150.
10. Manuel Carro and Manuel Hermenegildo. 2000. Tools for Constraint Visualisation: The VIFID/TRIFID Tool. In *Analysis and Visualization Tools for Constraint Programming: Constraint Debugging (Lecture Notes in Computer Science)*, Vol. 1870. Springer, 253–272.
11. Romuald Debruyne and Christian Bessière. 1997. Some Practicable Filtering Techniques for the Constraint Satisfaction Problem. In *Proceedings of the 15 th International Joint Conference on Artificial Intelligence*. 412–417.
12. Niklas Eén and Armin Biere. 2005. Effective Preprocessing in SAT Through Variable and Clause Elimination. In *Proc. of SAT 2005 (LNCS)*, Vol. 3569. Springer, 61–75.
13. Susan L. Epstein, Eugene C. Freuder, Richard M. Wallace, and Xingjian Li. 2005. Learning Propagation Policies. In *Second International Workshop on Constraint Propagation and Implementation, Volume I held in conjunction with CP 2005*. 1–15.
14. Eugene C. Freuder and Charles D. Elfe. 1996. Neighborhood Inverse Consistency Preprocessing. In *Proceedings of AAAI-96*. 202–208.
15. Eugene C. Freuder and Richard J. Wallace. 1991. Selective Relaxation For Constraint Satisfaction Problems. In *Proceedings of the IEEE Third International Conference on Tools with Artificial Intelligence*. 332–339.
16. Patrick J. Hayes. 1990. *Readings in Qualitative Reasoning About Physical Systems*. Morgan Kaufmann, Chapter The Second Naive Physics Manifesto, 46–63.
17. Matti Järvisalo, Marijn J. H. Heule, and Armin Biere. 2012. Inprocessing Rules. In *Automated Reasoning: 6th International Joint Conference (IJCAR 2012) (LNCS)*, Vol. 7364. Springer, 355–370.
18. Benjamin Kuipers. 1994. *Qualitative Reasoning - Modeling and Simulation with Incomplete Knowledge*. MIT Press.
19. Mikael Z. Lagerkvist and Christian Schulte. 2009. Propagator Groups. In *Proceedings of* 5 *th International Conference on Principle and Practice of Constraint Programming (CP'99) (LNCS)*, Vol. 5732. Springer, 524–538.
20. Alan K. Mackworth. 1977. Consistency in Networks of Relations. *Artificial Intelligence* 8 (1977), 99–118.
21. Mozilla. 2017. Using Web Workers. `https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API/Using_web_workers`. (2017). Accessed: 2017-04-21.
22. Mozilla and individual contributors. 2017. MDN Web Docs: Transferable. `https://developer.mozilla.org/en-US/docs/Web/API/Transferable`. (2017). Accessed: 2017-04-21.
23. Anastasia Paparrizou and Kostas Stergiou. 2012. Evaluating Simple Fully Automated Heuristics for Adaptive Constraint Propagation. In *Proceedings of the IEEE* 24 *th International Conference on Tools with Artificial Intelligence*. 880–885.
24. Anastasia Paparrizou and Kostas Stergiou. 2017. On Neighborhood Singleton Consistencies. In *Proceedings of the 26 th International Joint Conference on Artificial Intelligence*. 736–742.
25. Alfréd Rényi. 1961. *On Measures of Entropy and Information*. Technical Report. Hungarian Academy of Sciences, Budapest, Hungary.
26. Nikos Samaras and Kostas Stergiou. 2005. Binary Encodings of Non-binary Constraint Satisfaction Problems: Algorithms and Experimental Results. *JAIR* 24 (2005), 641–684.
27. Christian Schulte. 1996. Oz Explorer: A Visual Constraint Programming Tool. In *International Symposium on Programming Language Implementation and Logic Programming*. Springer, 477–478.
28. Christian Schulte, Guido Tack, and Mikael Z. Lagerkvist. 2015. *Modeling and Programming with Gecode*. `http://www.gecode.org/doc-latest/MPG.pdf`
29. Maxim Shishmarev, Christopher Mears, Guido Tack, and Maria Garcia de la Banda. 2016. Visual Search Tree Profiling. *Constraints* 21, 1 (2016), 77–94.
30. Helmut Simonis and Abder Aggoun. 2000. Search-Tree Visualisation. In *Analysis and Visualization Tools for Constraint Programming: Constraint Debugging (Lecture Notes in Computer Science)*, Vol. 1870. Springer, 191–208.
31. Helmut Simonis, Paul Davern, Jacob Feldman, Deepak Mehta, Luis Quesada, and Mats Carlsson. 2010. A Generic Visualization Platform for CP. In *Proceedings of* 16 *th International Conference on Principle and Practice of Constraint Programming (CP'10) (LNCS)*, Vol. 6308. Springer, 460–474.
32. Helmut Simonis and Barry O'Sullivan. 2011. Almost Square Packing. In *6697*, Vol. 6697. Springer, 196–209.
33. Kostas Stergiou. 2008. Heuristics for Dynamically Adapting Propagation. In *Proceedings of the 18 th European Conference on Artificial Intelligence*. 485–489.

34. Jordan Walke. 2017. React. `https://reactjs.org/`. (2017). Accessed: 2017-04-21.

35. Richard J. Wallace. 2015. SAC and Neighbourhood SAC. *AI Communications* 28, 2 (2015), 345–364.

36. Robert Woodward, Shant Karakashian, Berthe Y. Choueiry, and Christian Bessiere. 2011. Solving Difficult CSPs with Relational Neighborhood Inverse Consistency. In *Proceedings of AAAI-2011*. 112–119.

37. Robert J. Woodward. 2011. *Relational Neighborhood Inverse Consistency for Constraint Satisfaction: A Structure-Based Approach For Adjusting Consistency and Managing Propagation*. Master's thesis. Department of Computer Science and Engineering, University of Nebraska-Lincoln, Lincoln, NE.

38. Robert J. Woodward. 2018. *Higher-Level Consistencies: Where, When, and How Much*. Ph.D. Dissertation. University of Nebraska-Lincoln. Forthcoming.

39. Robert J. Woodward, Berthe Y. Choueiry, and Christian Bessiere. 2018. A Reactive Strategy for High-Level Consistency During Search. In *Proceedings of the 27 th International Joint Conference on Artificial Intelligence*. 1390–1397.

40. Robert J. Woodward, Shant Karakashian, Berthe Y. Choueiry, and Christian Bessiere. 2012. Revisiting Neighborhood Inverse Consistency on Binary CSPs. In *Proceedings of* 18 th *International Conference on Principle and Practice of Constraint Programming (CP'12) (Lecture Notes in Computer Science)*, Vol. 7514. Springer, 688–703.

41. Robert J. Woodward, Anthony Schneider, Berthe Y. Choueiry, and Christian Bessiere. 2014. Adaptive Parameterized Consistency for Non-Binary CSPs by Counting Supports. In *Proceedings of* 20 th *International Conference on Principle and Practice of Constraint Programming (CP'14) (LNCS)*, Vol. 8656. Springer, 755–764.

42. Andreas Wotzlaw, Alexander van der Grinten, and Ewald Speckenmeyer. 2013. *Effectiveness of Pre- and Inprocessing for CDCL-based SAT Solving*. Technical Report. Institut für Informatik, Universität zu Köln, Germany. `http://arxiv.org/abs/1310.4756`