A Structure-Based Variable Ordering Heuristic for CSPs Inspired by Visualization (A Preliminary Study)

Simon Schoenbeck[1], Berthe Y. Choueiry[1], Anastasia Paparrizou[2], and Christian Bessiere[2]

[1]Constraint Systems Laboratory, School of Computing, University of Nebraska–Lincoln, USA
[2]CNRS and University of Montpellier, France
Email: [1]{simonschoenbeck|berthe.choueiry}@gmail.com,
[2]{anastasia.paparrizou|bessiere}@lirrmm.fr

Abstract

Variable ordering highly impact the performance of backtrack search for solving Constraint Satisfaction Problems (CSPs), and remains the topic of extensive research. The heuristic `dwd` (and variations thereof) is currently acknowledged to provide the best results in general. This heuristic divides the domain size of a variable by its degree (i.e., number of constraints that apply to the variable) weighted by the number of conflicts in which the variable has appeared during search. We identify situations where the behavior of `dwd` degenerates and prevents search from terminating within an acceptable time limit. We design visualizations to illustrate the operation of `dwd` and reveal how it affects search performance. Then, we propose a new ordering heuristic (`mxClq`) that exploits the structure of the constraint network of the CSP, to guide `dwd` during search, and a criterion (Cluster Independence Ratio, `CIR`) to determine the applicability of `mxClq`. We empirically validate the effectiveness of our approach on benchmark problems.[1]

Keywords: Constraint Satisfaction, Search, Visualization.

[1]This report makes use of colors and is best read as a PDF or printed on a color printer.

1 Introduction

We claim that visualizations that summarize the behavior of the algorithms used to solve Constraint Satisfaction Problems (CSPs) provide useful explanations about the behavior of the algorithms and can effectively include the human user in the decision loop and in the problem-solving process. In this report, we provide evidence of our claim in the context of ordering the variables of a CSP for instantiation during backtrack search.

The performance of search depends heavily on the ordering of the variable during search. The variable ordering heuristic known as Dom/WDeg (`dwd`) [Boussemart *et al.*, 2004] is known to be most effective in practice. However, Howell *et al.* reported CSP instances on which `dwd` degenerates and is significantly outperformed by the usually less effective Dom/Deg (`dd`) [2020]. They introduced a heat-map based visualization to reflect the erratic behavior of `dwd` with respect to that of `dd`. Building on their observations, we introduce the following contributions:

1. A new visualization, Instantiation per Variable (`IpV`), relating the search behavior to the CSP's graphical structure.
2. A new variable ordering heuristic, MaxClique (`mxClq`), that exploits the structure of the constraint network.
3. A metric, Cluster Independence Ratio (`CIR`), to detect when a `mxClq` could be beneficial.
4. A preliminary empirical validation of our approach on benchmark problems.

This document is structured as follows. Section 2 reviews background information. Section 3 introduces the `IpV` visualization. Section 4 motivates and describes our new ordering heuristic, `mxClq`. Section 5 discusses the selection of the appropriate variable ordering heuristics to reduce runtime. Section 6 concludes with the current state of our research and future work.

2 Background and Related Work

Below we review background information.

2.1 Constraint Satisfaction Problems

Constraint Satisfaction Problems (CSPs) are used to model combinatorial decision problems. They are NP-complete by reduction from Boolean Satisfiability (SAT). The advantage of the CSP formulation is that it maintains the structure of the original problem and reflects the user understanding of the problem. A CSP is defined as follows:

1. Given a *constraint network*, which is:
    - A set $V = \{V_1, V_2, ..., V_n\}$ variables, to be assigned values from their respective domain $D_i = dom(V_i)$.
    - A set $C = \{C_1, C_2, ..., C_m\}$ of constraints that restrict the combinations of values that can be assigned to the variables at the same time. Each constraint $C_i : \langle scope(C_i), rel(C_i)\rangle$ is defined by its $scope(C_i) \subseteq (V)$ and $rel(C_i) \subseteq \prod_{x \in scope(C_i)}(dom(x))$. When the cardinality of the constraints scopes is no more than two, the CSP is called a *binary* CSP; otherwise, it is *nonbinary*.
2. The question is to determine whether there exists a solution to the CSP that assigns a value to each variable such that all constraints are simultaneously satisfied.

The graphical representation of a CSP, in which the *vertices* represent the variables and the *edges* represent the constraints and connect the variables in their scope, is called a *constraint (hyper) graph*. When the cardinality of a constraint scope is larger than two, replacing the *hyperedges* by cliques over the relevant vertices yields a graph called the *primal graph*. For a binary CSP, the primal graph and constraint graph are the same.

Other graphical representations of a CSP exist. We mention the dual graph and a tree decomposition of the primal graph. In the *dual graph* of a CSP, the vertices represent the constraints of the CSP and the edges link two vertices when the scopes of the corresponding constraints overlap. These edges denote equality constraints indicating that the variables in the scope intersection must be given the same values. A *tree decomposition* of a CSP is a tree embedding of the constraint network where the nodes of the tree are *clusters* of CSP variables and constraints with the following two conditions: (1) every constraint appears along with the variables in its scope in at least one node of the tree, and (2) all the nodes where a given variable appears form a connected subtree (i.e., the connectedness property). A common tree decomposition is obtained by triangulating the primal graph using the MinFill heuristic [Kjærulff, 1990; Dechter, 2003a], identifying the maximal cliques of the triangulated graph using the Maximal Cliques algorithm [Bron and Kerbosch, 1973], and building the tree decomposition using the Join Tree algorithm [Dechter, 2003b].

2.2 Variable Ordering Heuristics

Backtrack search is the only sound and complete algorithm for solving CSPs. The performance of search heavily depends on the instantiation order of the variables [Freuder, 1982; Purdom, 1983; Stone and Stone, 1987].

Let us consider the ideal case for a variable ordering with respect to a real-world single-thread computation model. For a satisfiable CSP with n variables, an ideal variable and value ordering makes exactly n assignments. In this paper, we only discuss variable ordering, and order values lexicographically. Thus, the ideal variable ordering is one that minimizes backtracks. For an unsatisfiable CSP instance, the ideal ordering selects variables from an unsatisfiable subproblem (ideally, a minimal conflict), and the size of the search tree is bounded by the product of the domain sizes of the variables in the conflict. The prevailing wisdom is to *instantiate the most constrained variable first*.

Some variable ordering heuristics exploit a structural property of the constraint graph, such as the width [Freuder, 1982], the maximal cardinality ordering [Tarjan and Yannakakis, 1984], the induced width or treewidth [Bertelè and Brioschi, 1972], the bandwidth [Zabih, 1990], or a tree decomposition [Jégou and Terrioux, 2003]. Such orderings aim at reducing backtracking effort. They are typically computed before search and maintained *static* throughout search.

However, *dynamic* variable ordering yields superior results. Indeed, after search instantiates a variable, the impact of this decision is propagated, by *lookahead*, on the remaining 'future' (i.e., unassigned) variables. Dynamic variable-ordering, combined with lookahead, is recognized to be essential for good search performance [Bessière and Régin, 1996].

Variable ordering remains the topic of active research investigations [Geelen, 1992; Michel and Van Hentenryck, 2012; Paparrizou and Wattez, 2020; Koriche *et al.*, 2022]. However, the heuristic known as Dom/WDeg (`dwd`) [Boussemart *et al.*, 2004] remains the basis of the most effective variations [Wattez *et al.*, 2019; Audemard *et al.*, 2023]. The following are some common variable ordering heuristics:

1. Lexicographic (`lex`) order, a static ordering, is usually used to break ties.
2. Least domain (`dom`), also called the least domain or minimum remaining values heuristic, orders variables based on their smallest remaining domain [Haralick and Elliott, 1980].
3. Degree (`deg`) chooses the variable connected to the largest number of future variables.
4. Dom/Deg (`dd`) chooses the variable with the smallest ratio of the domain size and degree, as defined above.
5. WDeg (`wdeg`), Dom/WDeg (`dwd`) increment the weights of the constraints that cause conflicts, which increases the weights of the degrees of the variables that appear in the scope of these constraints [Boussemart *et al.*, 2004].

2.3 Visualizations and Erratic Behavior of `dwd`

Our work builds on visualizations proposed by Woodward *et al.* where backtrack and lookahead effort are tracked and summarized per depth of the search tree [2018], and the ones proposed by Howell *et al.* where those metrics are sampled throughout the search and automatically organized into regimes of a history qualitatively describing the evolution of search over time [2020]. In particular, Howell et al. reported that solving some unsatisfiable graph-coloring instances of the `mug` benchmark[2] do not terminate within two hours when using `dwd` but terminate within minutes when using `dd` [2020]. To visualize the behavior of search, they proposed to display, as a heatmap, of the number of *variable instantiations per depth* `VIpD` of the search tree.

We choose the instance `mug88-1-3`, which is unsatisfiable, has 88 variables, 146 binary constraints, and all domains have size 3. Figure 1 shows our own rendering of their `VIpD` when solving `mug88-1-3`, with PREPEAK$^+$ (POAC) as lookahead and using `dd` and `dwd`. In the `VIpD`, the variables are ordered on the vertical axis by the weighted value of their instantiation depths [Howell *et al.*, 2020].[3] Variables instantiated at shallow search levels appear to the left of the

[2]Benchmark Library XCSP2.1 `https://www.cril.univ-artois.fr/~lecoutre/#/benchmarks`, accessed: 2025-04-11.

[3]Note that a variable whose domain is a singleton value is immediately instantiated, which is called the *domino effect* in lookahead. As a result, in dynamic ordering, such variables are instantiated before any other variables regardless of the ordering heuristic used.

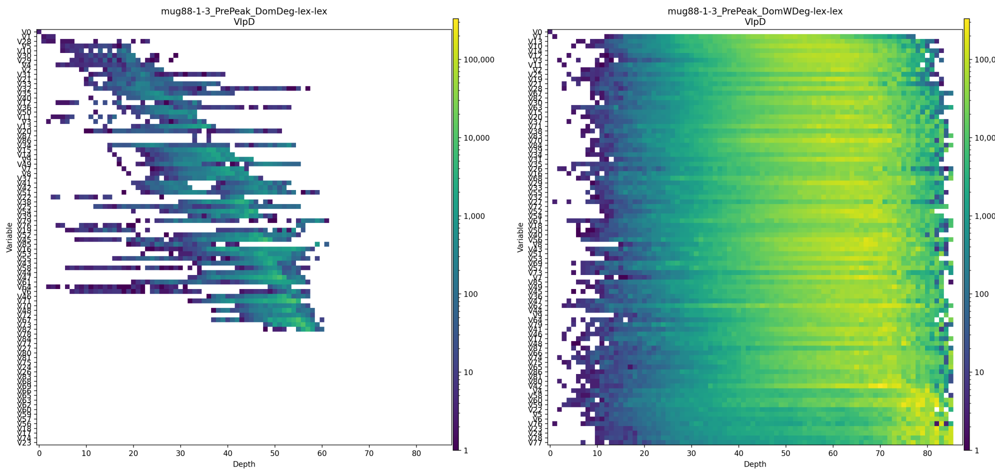

Figure 1: `VIpD` of search on `mug88-1-3`, using PrePeak$^+$ (POAC): `dd` (left) and `dwd` (right). These graphs use the same logarithmic color scale.

`VIpD`. Variables not instantiated by search are placed at the bottom of the vertical axis. Notice that a static variable ordering would produce a `VIpD` with dots (i.e., variable instantiations) along the diagonal. Values not on the diagonal display some change in the ordering of the variables. Dynamic variable orderings that are somehow stable would display most of the instantiations near the diagonal of the `VIpD`. However, when search is erratic (e.g., left of Figure 1), the `VIpD` reveals the variables are repeatedly instantiated at wide ranges of depths.

The `VIpD` of `dd` (Figure 1, left) shows no coloring in the bottom-right corner, which corresponds to deep levels of the search tree, indicating that no variable is instantiated at these depths. Thus, search terminates early, determining the instance to be unsatisfiable. In contrast, the `VIpD` of `dwd` (Figure 1, right) shows that search reached much deeper levels of the search tree and search made very many instantiations (notice the logarithmic vertical scale of the vertical axis). The `VIpD` reveals that `dwd` is unfocused and erratic, which explains the large CPU time difference between the two algorithms: `dwd` (right) takes more than 2,637 seconds to determine unsatisfiability whereas `dd` (left) terminates within 33 seconds. The analysis of this behavior was beyond the scope of the original paper [Howell *et al.*, 2020].

3 A New Visualization, `IpV`

We set out to study the erratic behavior of `dwd`. To this end, we display the number of *instantiations per variable* (`IpV`) directly on the primal graph of the CSP: This visualization shows us where search is active and relates its operations to the structure of the graph. Figure 2 shows the number of instantiations per variable `IpV` for the experiment of Figure 1.

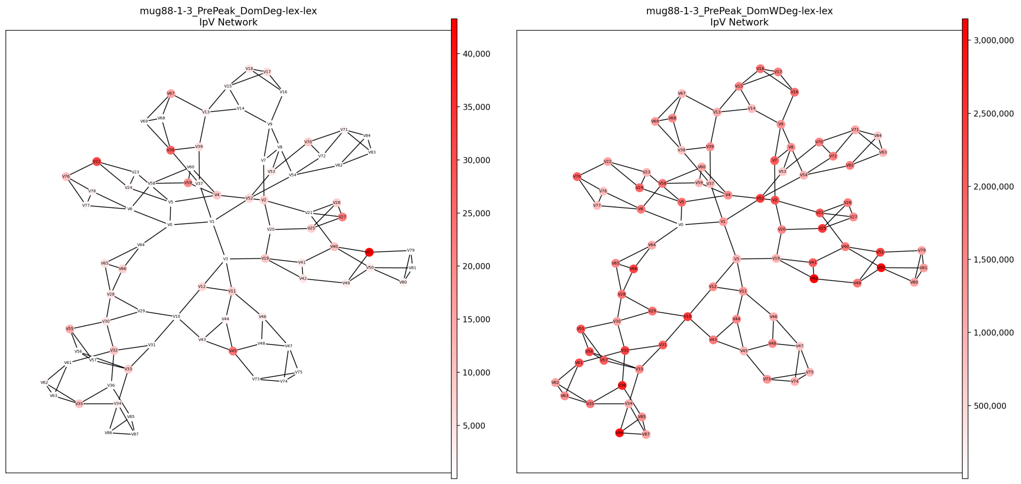

Figure 2: Instantiations per variable `IpV` for `mug88-1-3`, using PrePeak$^+$ (POAC) with `dd` (left) and `dwd` (right). The color scales are linear and are different to emphasize the colors in `dd`.

Comparison of `IpV` for `dwd` and `dd` in Figure 2 clearly shows that `dwd` is 'blindly jumping around' and whereas `dd` is focused on a small number of variables. By restricting search to relatively a small number of variables, we conjecture that `dd` is able to exhaustively explore all combinations of values for these variables to determine that the instance is unsatisfiable.

4 A New Variable Ordering Heuristic

Examining the `IpV` of `dd` (Figure 2, left) reveals that search focuses on 'key' vertices that seem to connect the various components of the graph. This fact inspired us to prioritize the instantiation of variables that appear in the largest number of maximal cliques in a triangulation of the primal graph, which correspond to the largest number of clusters in the corresponding tree decomposition, computed as described in Section 2.1.

4.1 Maximal Cliques Ordering, `mxClq`

Our new ordering heuristic, Maximal Cliques (`mxClq`), chooses to instantiate first the variable that appears in the largest number of maximal cliques given some triangulation of the primal graph of the CSP (Section 2.1). Ties are broken using another heuristic (e.g., `lex` or `dwd`) to be specified. The number of maximal cliques in which a variable appears is computed before the search starts and is not updated during the search.

We do not advocate replacing `dwd` with `mxClq`. In fact, `dwd`, and its variations, is the current de facto standard. Furthermore, `mxClq` gives a static ordering whereas `dwd` gives a dynamic ordering, which is known to be essential for good search performance [Bessière and Régin, 1996]. Because `dwd` can sometimes lose focus and become erratic, as in the case of unsatisfiable `mug` instances, we propose to use `mxClq` to force `dwd` to focus its operations on the variables appearing in the largest number of cliques. Consequently, we first apply `mxClq` to identify the potentially critical variables, and then apply `dwd` to this restricted set, breaking ties lexicographically. As a result, `mxClq` forces `dwd` to prioritize the instantiation of critical variables.

4.2 Applying `mxClq` to `mug88-1-3`

Figure 3 shows how `mxClq` is able to focus search on `mug88-1-3` and significantly improves search performance. Indeed, `mxClq` determined unsatisfiability in less 10 seconds, in contrast to about 33 seconds for `dd`, and more than 43 minutes for `dwd`. We clearly see that `mxClq`:

1. terminates search at earlier depth levels than `dwd`;
2. focuses search more ‘sharply’ than `dd` and `dwd`; and
3. requires much fewer instantiations than `dd` and `dwd`.

Thus, on this particular instance, `mxClq` seems to benefit from exploiting the structure of the constraint network (via the maximal cliques) and also from the ability of `dwd` to prioritize bottlenecks.

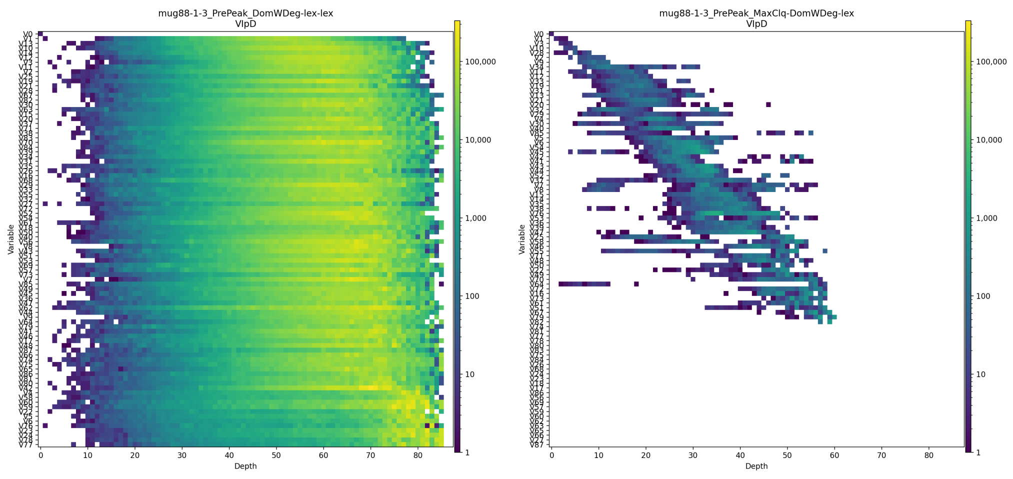

Figure 3: `VIpD` of search on `mug88-1-3`, using PREPEAK$^+$ (POAC): `dwd` (left) and `mxClq` (right). These graphs use the same logarithmic color scale as Figure 1.

Our `IpV` visualization, Figure 4, clearly shows how `mxClq` (right) focuses the search on select nodes in comparison to `dwd` (left).

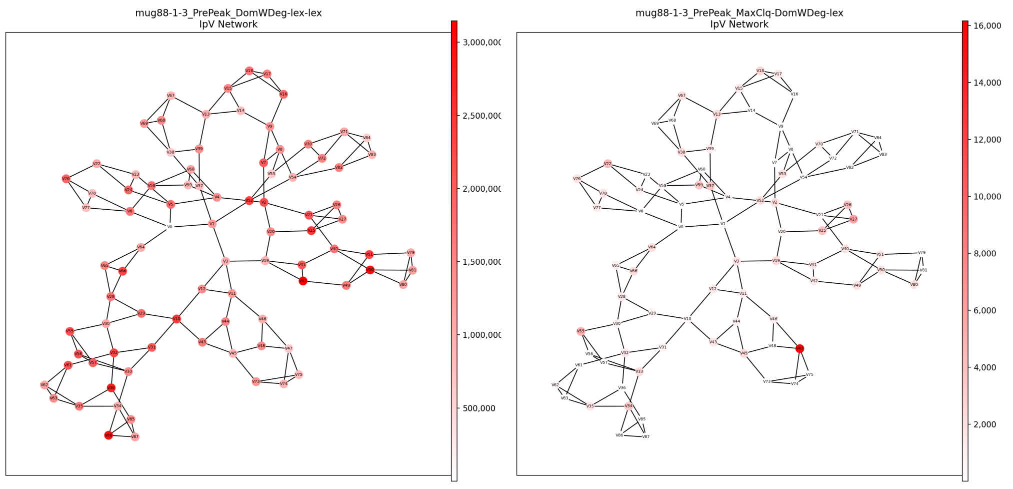

Figure 4: Instantiations per variable (`IpV`) for `mug88-1-3` with PREPEAK$^+$ (POAC), using `dwd` (left), `mxClq` (right). The linear color scales are different to emphasize the colors in `mxClq`.

4.3 Forced Static Ordering on `mug88-1-3`

One natural question is to analyze the variables on which the heuristics `dd`, `dwd` (the current standard), and `mxClq` are 'chalking.' To this end, we sort the variables of `mug88-1-3` in the decreasing value of their number of instantiations by each of `dd`, `dwd`, and `mxClq`. The top ranking variables appear as the most critical for each of the three heuristics. Then, we force each resulting ordering as a *forced static* ordering for search in order to assess the 'discriminating' effectiveness of each heuristic. The corresponding `VIpD` are shown in Figure 5.[4]

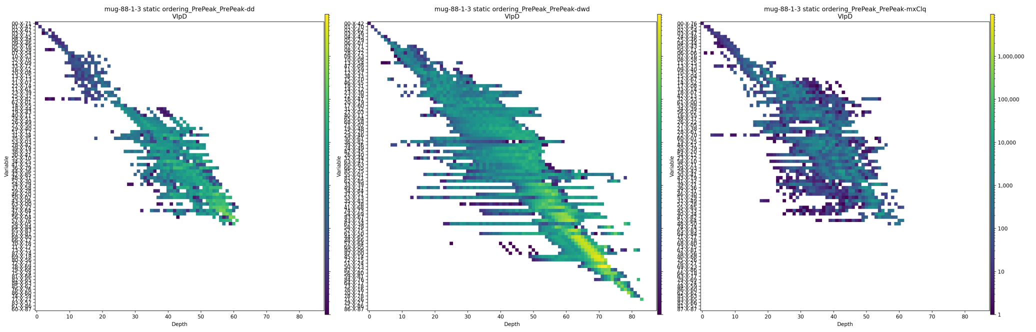

Figure 5: `VIpD` of forced static ordering on `mug88-1-3`, using PrePeak$^+$ (POAC): `dd` (left), `dwd` (middle) and `mxClq` (right) in the same logarithmic color scale.

Table 1 reports the performance of search in dynamic and forced static orderings in terms of CPU time, number of backtracks (#BT), number of nodes visited (#NV), and the deepest level reached to determine unsatisfiability.

Table 1: Comparing performance of dynamic and forced static variable ordering on `mug88-1-3` (88 variables) using PrePeak$^+$ (POAC).

	Dynamic ordering			Forced static ordering		
	`dd`	`dwd`	`mxClq`	`dd`	`dwd`	`mxClq`
CPU [sec]	35.41	2,833.28	**10.56**	53.13	410.35	49.61
Total #BT	246,781	102,846,409	**75,947**	635,373	5,055,471	413,776
Total #NV	409,796	130,713,508	**120,194**	853,576	7,203,069	581,470
Max depth	61	85	**60**	61	83	61

[4]The vertical-axis scales in Figures 3 and 5 are different.

Comparing the results in Table 1, and the VIpD in Figures 1 and 3 (dynamic ordering) with the corresponding plots in Figure 5 (forced static ordering), we notice that:

1. The forced static ordering deteriorates the search performance for both dd and mxClq, which hints to the difficulty of isolating the conflicts in this instance.

2. It improves the search performance for dwd, which shows how misguided dwd can be.

This study raises new questions:

1. Can mxClq 'uncover' a good variable ordering on a *satisfiable* instance?

2. How does mxClq perform on other benchmark classes, particularly those whose primal graph may not be structured?

3. Can we predict when mxClq would be advantageous?

4. Can we use the static ordering corresponding to mxClq to characterize the minimal conflicts of unsatisfiable instances, which is a known NP-hard problem?

In Section 4.4 we repeat the study above on a satisfiable problem instance, namely, fpga-10-9. In Section 4.5, we report benchmark on which mxClq outperforms dwd and dd. In Section 5, we propose a criterion, CIR, based on the structure of a tree decomposition of the primal graph in order to predict the appropriateness of mxClq. The last question is beyond the scope of this report.

4.4 Forced Static Ordering on fpga-10-9

fpga-10-9 is a satisfiable CSP with 135 variables, domain of size 2, 118 non-binary constraints (with arity varying from 5 to 10) whose primal graph has a density of about 15%. Table 2 shows the search performance using PREPEAK$^+$ (POAC), with dd, dwd, and mxClq for dynamic variable ordering and the corresponding forced static ordering. The corresponding VIpD are shown in Figure 6.

We note that, unlike mug88-1-3, dd fails to terminate within 2 hours. dwd finds a solution after more than 283 seconds, but mxClq terminates successfully

Table 2: Comparing performance of dynamic and forced static variable ordering on `fpga-10-9` (135 variables) using PREPEAK$^+$ (POAC).

	Dynamic ordering			**Forced static ordering**		
	`dd`	`dwd`	`mxClq`	`dd`	`dwd`	`mxClq`
CPU [sec]	>7,200.00	283.80	0.24	>7,200.00	1,985.09	**0.07**
Total #BT	>515,850,836	6,209,526	2,366	>261,621,797	6,980,745	**0**
Total #NV	>634,219,575	8,135,467	3,058	>342,221,921	10,554,376	**135**
Max depth	128	**135**	**135**	120	**135**	**135**

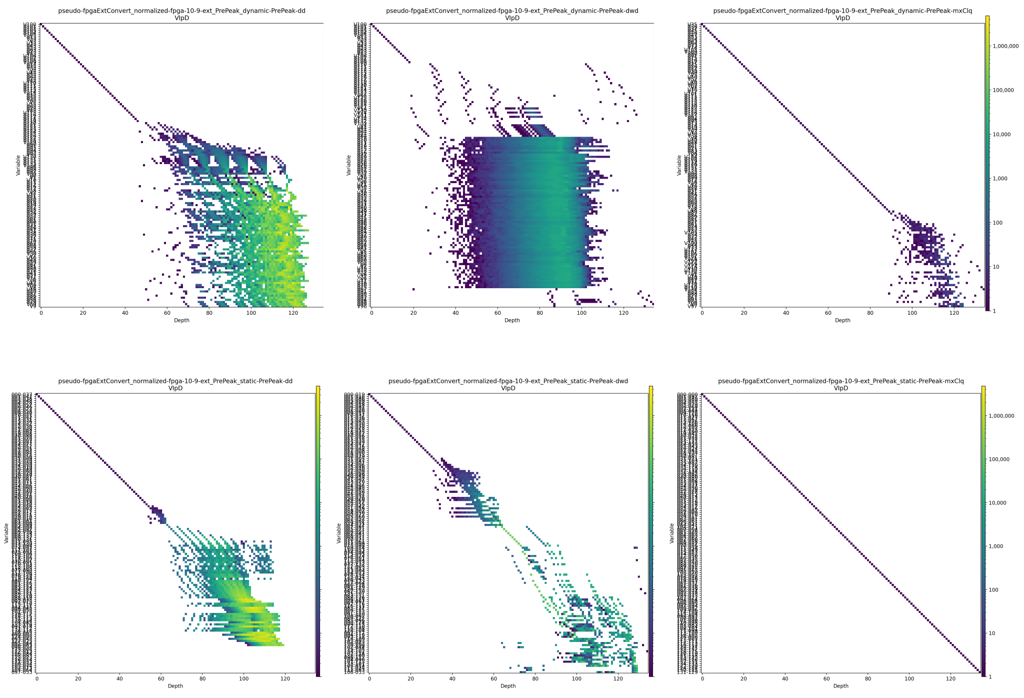

Figure 6: Variable instantiations per depth (`VIpD`) on `fpga-10-9`, using PREPEAK$^+$ (POAC) for lookahead: `dd` (left), `dwd` (center), and `mxClq` (right). The top row shows the graphs for dynamic variable ordering and the bottow row shows the corresponding forced static orderings. All graphs have the same logarithmic color scale. Notice the forced static ordering found by `mxClq` solves the instance in a backtrack-free manner.

in 0.24 seconds. The `VIpD`'s in Figure 6 show how `dd` and `dwd` both lack focus whereas `mxClq` seems to proceed towards the solution with relatively little hesitation.

Most notably, with the forced variable ordering, the search performance of `dd` and `dwd` deteriorates, whereas `mxClq` proceeds in a backtrack-free manner, thus hinting to its ability to find a 'perfect' variable ordering. An interesting open question is how to exploit such a behavior for solving CSPs.

4.5 Validating `mxClq` on Difficult Problems

Preliminary experiments on benchmark problems showed that `mxClq` outperforms `dd` and `dwd` on difficult 'structured' problems. What constitutes a 'structured' instance remains to be defined; `CIR` is our first attempt to this end.

Table 3 summarizes these results per benchmark, with the details of the results per instance reported in Table 4 in Appendix B.

Table 3: Summary on select benchmark with positive results. Detailed results are included as an appendix.

			#Solved			**CPU Time [sec]**	
Benchmark	#Instances	`dd`	`dwd`	`mxClq`	`dd`	`dwd`	`mxClq`
STR2							
graphColoring-mug	8	4	4	4	>28,800.09	>28,800.09	>28,800.10
jobShop-e0ddr1	10	4	5	**6**	>43,230.58	>37,858.63	**>28,845.27**
jobShop-e0ddr2	10	4	5	**8**	>43,240.88	>43,240.89	**>14,478.76**
jobShop-enddr2	6	3	3	**5**	>21,633.93	>21,633.93	**>7,254.14**
pseudo-fpga	21	0	3	**20**	>151,200.00	>135,651.56	**>7,851.66**
APOAC							
graphColoring-mug	8	7	6	**8**	>9,424.86	>16,402.89	**5,879.48**
jobShop-e0ddr1	10	4	4	**7**	>47,398.80	>47,399.02	**>28,850.98**
jobShop-e0ddr2	10	4	4	**8**	>49,887.98	>49,888.22	**>25,401.54**
jobShop-enddr2	6	3	3	**6**	>27,238.25	>27,238.45	**12,294.97**
pseudo-fpga	21	0	3	**20**	>151,200.00	>13,9962.53	**>8,147.67**
PrePeak$^+$ with POAC							
graphColoring-mug	8	**8**	7	**8**	465.14	>17,715.27	**65.90**
jobShop-e0ddr1	10	4	5	**7**	>43,226.69	>38,435.73	**>21,886.81**
jobShop-e0ddr2	10	4	4	**8**	>43,263.27	>43,263.27	**>14,521.47**
jobShop-enddr2	6	3	3	**6**	>21,653.32	>21,653.33	**282.68**
pseudo-fpga	21	0	3	**20**	>151,200.00	>135,977.42	**>7,471.40**

The discussed benchmark include both satisfiable and unsatisfiable instances, binary and nonbinary CSPs, with domain sizes small and large. In Appendix A, we include one representative primal graph of each benchmark. We also include the corresponding dual graph, for additional clarity.

Each row in Table 3 gives the name of the benchmark, the total number of instances in the benchmark, and summarizes the performance of three lookahead schemas, namely, STR2 [Lecoutre, 2011], APOAC [Balafrej *et al.*, 2014], and PREPEAK$^+$ (POAC) [Woodward *et al.*, 2018]) for each of `dd`, `dwd`, and `mxClq`. The table shows the number of instances solved by each algorithm within two hours, and the cumulated CPU time in seconds, where the character $>$ indicates that the time reported is a lower bound. For a given lookahead schema, the best result of the three ordering heuristics is formatted in boldface. The results in Tables 3 and 4 validate the promise of `mxClq`.

5 Predicting Appropriateness of **`mxClq`**

We designed `mxClq` to 'guide' `dwd` by exploiting the structure of the primal graph. There is no reason to believe that `mxClq` will perform well on all CSPs. Clearly, when the clusters of a tree decomposition of the primal graph overlap on most of their variables and the same variables appear in most clusters, `mxClq` would hardly seem appropriate. We hypothesize that a metric that measures the 'independence' of non-adjacent clusters of a tree decomposition may identify the instances where `mxClq` is beneficial. By independence, we mean that non-adjacent clusters share no variables. The Cluster Independence Ratio (`CIR`) is computed from a tree decomposition of a CSP. A variable typically appears in many clusters of a tree decomposition because of the connectedness property. By construction, two adjacent clusters share one or more variables, which form the separator between the two clusters. Non-adjacent clusters may or may not have variables in common. We define the Cluster Independence Ratio (`CIR`) as the ratio of the number of pairs of non-adjacent clusters with no common variables to the total number of pairs of non-adjacent clusters, it ranges in [0,1]

$$\texttt{CIR} = \frac{\text{\# non-adjacent cluster pairs with no variables in common}}{\text{\# non-adjacent cluster pairs}}$$

When `CIR` =0, every pair of non-adjacent clusters has a variable in common; when `CIR` =1, all non-adjacent clusters have no variables in common. The `CIR` is applicable only to trees with more than two clusters: trees with two clusters have no non-adjacent clusters.

Based on the observation that `mug` instances have a high `CIR`, we expected that `mxClq` to perform well on problems with high `CIR` values, e.g., larger than 0.5. However, Table 4 shows that, while `mug` is included by this threshold, all the other listed benchmark are not, yet they benefit from `mxClq`. Clearly, we need more problem features as 'indicators' of the appropriateness of `mxClq` in order to eventually automate the selection of an ordering heuristic.

Towards identifying useful metrics, we classified, using decision trees of limited depth, about 2,000 instances on which search with both `dwd` and `mxClq` completed within two hours. As training data, we used the runtime performance of `dwd` and `mxClq`, a large set of CSP features [Geschwender *et al.*, 2016], our new metrics (`CIR`), and the number of clusters per variable. We limited the depth of the decision trees to prevent overfitting and maintain the 'interpretability' of the trees by humans.

To train decision trees, we labeled instances as class 1 when `mxClq` is at least a minute faster than `dwd`, class -1 when `dwd` is at least a minute faster than `mxClq`, and class 0 otherwise. Ignoring the class 0 instances focuses the training of the decision tree on the extreme cases where a lot of time can be saved or lost, thus providing a buffer where the decision tree could select either variable ordering and not be penalized. We then trained, with subsets of the selected features, several decision tree on all instances of classes 1 and -1.

We recognize, from the outset, that using graph-structure metrics is not sufficient. Indeed, `mug88-1-3` is a member of class 1, whereas `mug88-1-4` is a member of class 0. However, these two instances differ only by the size of their domain, namely, three instead of four, see Table 4. This pair of instances illustrates the limit of using graph structures as metrics for runtime classification. Indeed, two instances with almost identical metrics may have significantly different run times, which is problematic for training classifiers. Other approaches, orthogonal to static classifiers, such as probing and restarting search are worth considering [Habet and Terrioux, 2019; Paparrizou and Wattez, 2020].

Our preliminary study identified the following features as salient for favoring `mxClq` over `dwd`: covariance of degree of the primal graph, number of cliques of a tree decomposition of the primal graph, number of *interacting* cluster pairs of this tree decomposition (where we define interacting pairs as non-adjacent cluster pairs with at least one common variable), number of constraints in the CSP, `CIR`, density of the primal graph, and clustering coefficient of the primal graph.

The other metrics relate to the structure of the primal and tree decomposition. Degree covariance seems reasonable for determining when to use `mxClq`. A graph where the degree covariance is low means that the degrees for all variables are similar values. This fact, in combination with a low covariance in domain size, could cause `dwd` to similarly weigh all of the variables.

Our current results show that, while the aforementioned metrics are relevant for identifying the applicability of `mxClq`, this classification task is difficult due to small differences in graph structure and also in the CSP itself, which have significant impact on the performance of search. Further work is needed to assess the benefits of using a classifier to select an ordering heuristics before search.

6 Conclusions

In this report, we show how visualization supports the understanding of search behavior and the design of new techniques in the context of solving CSPs with search. While the `VIpD` allowed the detection of the erratic behavior of search on the over-constrained `mug` instances, our `IpV` inspired the design of a new ordering heuristic (`mxClq`) and a new graph-structure metric (`CIR`) towards the development of structure-based selection strategies of ordering heuristics.

One should not ignore the additional cost incurred for computing the various metrics (e.g., building a tree decomposition of a graph), which may be non-negligible. However, given the significant impact on the performance of search (whose cost is exponential in the number of variables), such additional effort (whose cost is polynomial) would ideally still save time, overall. Our current ability to assess the benefits of a given heuristic is limited by the currently identified metrics, the tested variable ordering heuristics, and the inherent complexity of NP-hard problems.

We envision two main directions for future work. The first direction is concerned with the design of new visualizations such as enriching the `IpV` with information about how lookahead affects domain wipe-outs and constraint and variable weights.The second direction is to extend our study to include more diverse ordering heuristics, such as the ones listed in Section 2.2.

Acknowledgments

This work was completed utilizing the Holland Computing Center of the University of Nebraska, which receives support from the Nebraska Research Initiative. The authors acknowledge the help of Avery Penn in the implementation of the visualizations tool, and that of Robert Woodward and Anthony Schneider for their guidance in extending the Constraint Systems Laboratory's constraint solver, STAMPEDE.

References

[Audemard *et al.*, 2023] Gilles Audemard, Christophe Lecoutre, and Charles Prud'Homme. Guiding Backtrack Search by Tracking Variables During Constraint Propagation. In *International Conference on Principles and Practice of Constraint Programming (CP'23)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2023.

[Balafrej *et al.*, 2014] Amine Balafrej, Christian Bessiere, El-Houssine Bouyakhf, and Gilles Trombettoni. Adaptive Singleton-Based Consistencies. In *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence (AAAI-14)*, pages 2601–2607, 2014.

[Bertelè and Brioschi, 1972] Umberto Bertelè and Francesco Brioschi. *On Non-Serial Dynamic Programming*. Academic Press, 1972.

[Bessière and Régin, 1996] Christian Bessière and Jean-Charles Régin. MAC and Combined Heuristics: Two Reasons to Forsake FC (and CBJ?) on Hard Problems. In *Proceedings of the Second International Conference on Principle and Practice of Constraint Programming (CP-96)*, volume 1118 of *LNCS*, pages 61–75. Springer, 1996.

[Boussemart *et al.*, 2004] Frédéric Boussemart, Fred Hemery, Christophe Lecoutre, and Lakhdar Sais. Boosting Systematic Search by Weighting Constraints. In *Proceedings of the Seventeenth European Conference on Artificial Intelligence (ECAI-06)*, pages 146–150, 2004.

[Bron and Kerbosch, 1973] Coen Bron and Joep Kerbosch. Algorithm 457: Finding All Cliques of an Undirected Graph. *Communications of the ACM*, 16(9):575–577, September 1973.

[Dechter, 2003a] Rina Dechter. *Constraint Processing*, chapter Directional Consistency, page 89. Morgan Kaufmann, 2003.

[Dechter, 2003b] Rina Dechter. *Constraint Processing*, chapter Directional Consistency, page 90. Morgan Kaufmann, 2003.

[Freuder, 1982] Eugene C. Freuder. A Sufficient Condition for Backtrack-Free Search. *JACM*, 29 (1):24–32, 1982.

[Geelen, 1992] Pieter Andreas Geelen. Dual Viewpoint Heuristics for Binary Constraint Satisfaction Problems. In *Proceedings of the Tenth European Conference on Artificial Intelligence (ECAI-92)*, pages 31–35, Vienna, Austria, 1992.

[Geschwender *et al.*, 2016] Daniel J. Geschwender, Robert J. Woodward, Berthe Y. Choueiry, and Stephen D. Scott. A Portfolio Approach for Enforcing Minimality in a Tree Decomposition. Technical Report TR-UNL-CSE-2016-0003, Department of Computer Science and Engineering, University of Nebraska-Lincoln, Lincoln, NE, 2016.

[Habet and Terrioux, 2019] Djamal Habet and Cyril Terrioux. Conflict History Based Search for Constraint Satisfaction Problem. In *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*, SAC '19, page 1117–1122, New York, NY, USA, 2019. Association for Computing Machinery.

[Haralick and Elliott, 1980] Robert M. Haralick and Gordon L. Elliott. Increasing Tree Search Efficiency for Constraint Satisfaction Problems. *Artificial Intelligence*, 14:263–313, 1980.

[Howell *et al.*, 2020] Ian S. Howell, Berthe Y. Choueiry, and Hongfeng Yu. Visualizations to Summarize Search Behavior. In Helmut Simonis, editor, *Proceedings of the 26th International Conference on Principles and Practice of Constraint Programming*, pages 392–409, 2020.

[Jégou and Terrioux, 2003] Philippe Jégou and Cyril Terrioux. Hybrid Backtracking Bounded by Tree-Decomposition of Constraint Networks. *Artificial Intelligence*, 146:43–75, 2003.

[Kjærulff, 1990] U. Kjærulff. Triagulation of Graphs - Algorithms Giving Small Total State Space, 1990.

[Koriche *et al.*, 2022] Frederic Koriche, Christophe Lecoutre, Anastasia Paparrizou, and Hugues Wattez. Best Heuristic Identification for Constraint Satisfaction. In *Proceedings of the Thirty-First International Joint Conference on Artificial Intelligence, IJCAI-22*, pages 1859–1865, 7 2022.

[Lecoutre, 2011] Christophe Lecoutre. STR2: Optimized Simple Tabular Reduction for Table Constraints. *Constraints*, 16(4):341–371, 2011.

[Michel and Van Hentenryck, 2012] Laurent Michel and Pascal Van Hentenryck. Activity-Based Search for Black-Box Constraint Programming Solvers. In *Proc. of CPAIOR 2012*, volume 7298, pages 228–243. Spring, 2012.

[Paparrizou and Wattez, 2020] Anastasia Paparrizou and Hugues Wattez. Perturbing Branching Heuristics in Constraint Solving. In *Lecture Notes in Computer Science*, volume 12333 of *Lecture Notes in Computer Science*, Louvain-la-Neuve, Belgium, September 2020.

[Purdom, 1983] Paul Walton Purdom. Search Rearrangement Backtracking and Polynomial Average Time. *Artificial Intelligence*, 21(1):117–133, 1983. Search and Heuristics.

[Stone and Stone, 1987] Harold S. Stone and Janice M. Stone. Efficient Search Techniques—An Empirical Study of the N-Queens Problem. *IBM Journal of Research and Development*, 31(4):464–474, 1987.

[Tarjan and Yannakakis, 1984] Robert Endre Tarjan and Mihalis Yannakakis. Simple Linear-Time Algorithms to Test Chordality of Graphs, Test Acyclicity of Hypergraphs, and Selectively Reduce Acyclic Hypergraphs. *SIAM Journal on Computing*, 13(3):566–579, 1984.

[Wattez *et al.*, 2019] Hugues Wattez, Christophe Lecoutre, Anastasia Paparrizou, and Sébastien Tabary. Refining Constraint Weighting. In *2019 IEEE 31st International Conference on Tools with Artificial Intelligence (ICTAI)*, pages 71–77, 2019.

[Woodward *et al.*, 2018] Robert J. Woodward, Berthe Y. Choueiry, and Christian Bessiere. A Reactive Strategy for High-Level Consistency During Search. In *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence (IJCAI-18)*, pages 1390–1397, 2018.

[Zabih, 1990] Ramin Zabih. Some Applications of Graph Bandwidth to Constraint Satisfaction Problems. In *Proceedings of the Eighth National Conference on Artificial Intelligence (AAAI-90)*, pages 46–51, Boston, MA, 1990.

A Appendix - Graph Structure Visualizations

The following figures are examples of the network visualizations we implemented. The primal graphs were constructed using Python 3.12 and the NetworkX library. This setup was also used to produce the earlier `IpV` and the `VIpD` figures. The dual graphs were constructed using JavaScript and D3.js library. This setup allows for more user interactivity such as moving nodes and changing colors of the graphs in the browser.

Each figure is a representative instance of their benchmark. For the `mug` instances in particular, the graphs for instance that only differ by the final number, (e.g `mug88-1-3` and `mug88-1-4`) are identical except for the domain size. Figure 7 shows the primal and dual graphs for `mug88-1-3`, and identical to the graphs for `mug88-1-4`, with 88 variables and 146 constraints. All `jobShop` instances have 50 variables and 265 constraints, which produce similar graphs: `e0ddr1-10-by-5-1` Figure 8, `e0ddr2-10-by-5-1` Figure 9, `enddr2-10-by-5-10` Figure 10. The `mug` and `jobShop` instances are all binary so every constraint is represented by a single edge. The visualization of `fpga-10-8` Figure 11 is more visually dense than the others because it has 120 variables the maximum arity is 10. With a non-binary graph each constraint requires an edge between every pair of variables in the scope.

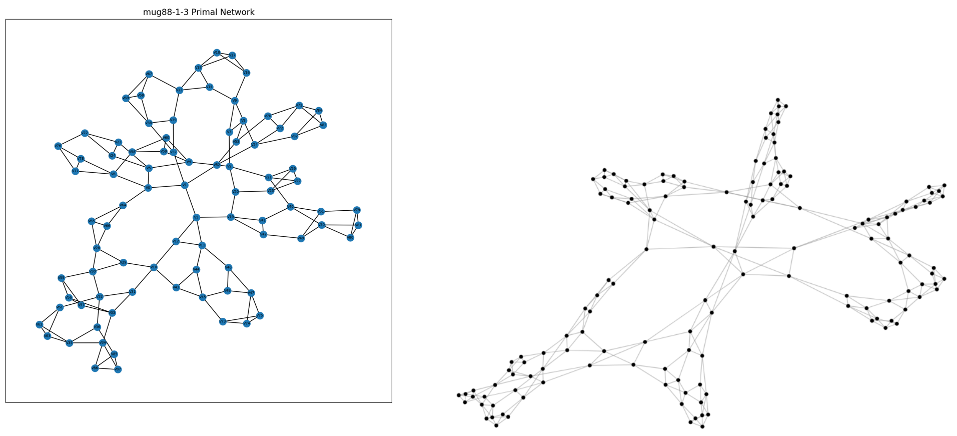

Figure 7: `mug88-1-3` primal graph (left) and dual graph (right)

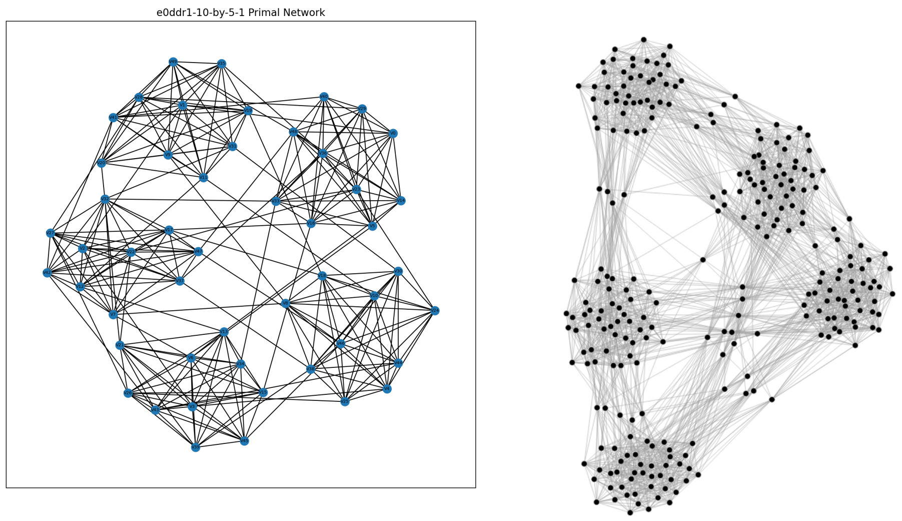

Figure 8: `e0ddr1-10-by-5-1` primal graph (left) and dual graph (right)

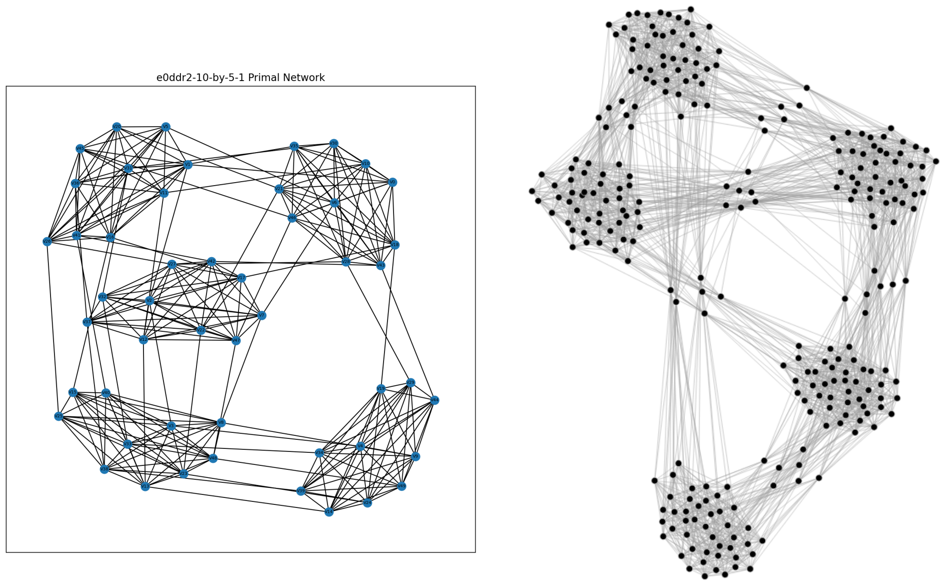

Figure 9: `e0ddr2-10-by-5-1` primal graph (left) and dual graph (right)

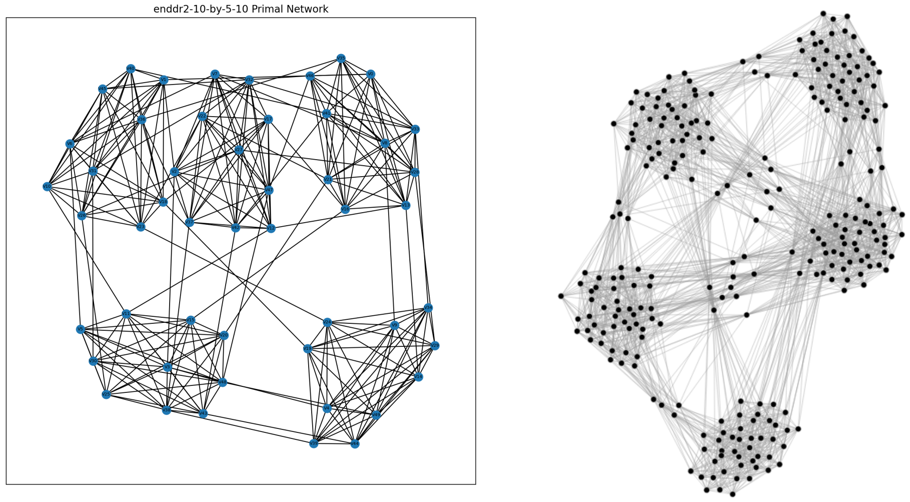

Figure 10: `enddr2-10-by-5-10` primal graph (left) and dual graph (right)

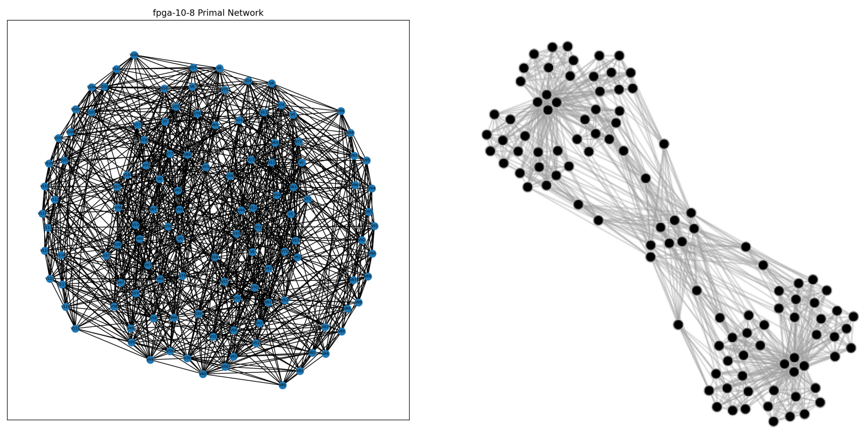

Figure 11: `fpga-10-8` primal graph (left), dual graph (right)

B Appendix - Detailed Benchmark Performance

Each row in Table 4 contains: the name of the benchmark, name of the instance, structural metric about the CSP and the Tree Decomposition, and the Runtime in seconds. The structural metrics are: number of constraints, number of variables, maximum arity of the constraints, maximum domain size, number clique in the tree decomposition (#clq), `CIR`, and the average, and coefficient of variation of the number of cliques per variable (#clq/-var). The satisfiability is noted with 0, 1, and - for unsatisfiable, satisfiable, and no algorithm terminated, respectively. The runtime performance is shown in seconds for three lookahead schemas, namely, STR2 [Lecoutre, 2011], APOAC [Balafrej *et al.*, 2014], and PREPEAK$^+$ (POAC) [Woodward *et al.*, 2018]) for each of `dd`, `dwd`, and `mxClq`. For instances where the algorithm did not terminate withinr two hours a $> 7,200$ indicates that the time reported is a lower bound. For a given lookahead schema, the best result of the three ordering heuristics is formatted in boldface. The total rows for each benchmark report the cumulative time for all runs and are summarized in Table 3.

Table 4: Performance per instance

							Tree Decomposition				**CPU Time [sec]**								
											STR2			**APOAC**			**PrePeak + (POAC)**		
Benchmark	Instance	#Constraints	# Variables	Max Arity	Max Dom. Size	Satisfiable	**#clq**	**CIR**	**#clq/var** Avg	**#clq/var** CoV	`dd`	`dwd`	`mxClq`	`dd`	`dwd`	`mxClq`	`dd`	`dwd`	`mxClq`
graphColoring-mug	mug100-1-3	166	100	2	3	0	65	94%	2.280	6%	>7,200.00	>7,200.00	>7,200.00	2,009.75	>7,200.00	390.42	127.78	>7,200.00	36.16
	mug100-1-4	166	100	2	4	1	65	94%	2.280	6%	0.02	0.02	0.03	0.08	0.08	0.08	0.03	0.03	0.03
	mug100-25-3	166	100	2	3	0	65	94%	2.280	6%	>7,200.00	>7,200.00	>7,200.00	>7,200.00	>7,200.00	5,344.05	292.02	>7,200.00	9.26
	mug100-25-4	166	100	2	4	1	65	94%	2.280	6%	0.02	0.02	0.03	0.08	0.08	0.08	0.03	0.03	0.03
	mug88-1-3	146	88	2	3	0	57	94%	2.271	13%	>7,200.00	>7,200.00	>7,200.00	100.83	1,553.03	70.82	35.41	2,833.28	10.56
	mug88-1-4	146	88	2	4	1	57	94%	2.271	13%	0.02	0.02	0.02	0.06	0.06	0.07	0.03	0.03	0.03
	mug88-25-3	146	88	2	3	0	57	93%	2.271	6%	>7,200.00	>7,200.00	>7,200.00	114.00	449.58	73.89	9.82	481.87	9.80
	mug88-25-4	146	88	2	4	1	57	93%	2.271	6%	0.02	0.02	0.02	0.06	0.06	0.07	0.03	0.03	0.03
	Total										>28,800.09	>28,800.09	>28,800.10	>9,424.86	>16,402.89	5,879.48	465.14	>17,715.27	65.90
jobShop-e0ddr1	e0ddr1-10-by-5-1	265	50	2	115	1	461	7%	8.420	7%	>7,200.00	>7,200.00	7.23	>7,200.00	>7,200.00	964.55	>7,200.00	>7,200.00	10.6
	e0ddr1-10-by-5-2	265	50	2	118	1	461	7%	8.420	4%	>7,200.00	>7,200.00	>7,200.00	>7,200.00	>7,200.00	1,144.91	>7,200.00	>7,200.00	181.01
	e0ddr1-10-by-5-3	265	50	2	114	-	432	7%	8.240	7%	>7,200.00	>7,200.00	>7,200.00	>7,200.00	>7,200.00	>7,200.00	>7,200.00	>7,200.00	>7,200.00
	e0ddr1-10-by-5-4	265	50	2	116	1	482	9%	8.540	6%	>7,200.00	1,828.04	7.29	>7,200.00	>7,200.00	975.22	>7,200.00	1,951.06	10.66
	e0ddr1-10-by-5-5	265	50	2	120	1	451	9%	8.420	7%	7.59	7.59	7.56	1,072.61	1,072.66	940.68	11.38	11.38	11.36
	e0ddr1-10-by-5-6	265	50	2	124	1	398	9%	7.960	7%	7.94	7.94	7.90	1,077.83	1,077.89	1,039.85	11.99	11.99	11.95
	e0ddr1-10-by-5-7	265	50	2	118	1	611	3%	9.100	6%	7.31	7.32	7.39	998.23	998.28	1,085.08	10.77	10.77	10.84
	e0ddr1-10-by-5-8	265	50	2	121	-	479	3%	8.420	7%	>7,200.00	>7,200.00	>7,200.00	>7,200.00	>7,200.00	>7,200.00	>7,200.00	>7,200.00	>7,200.00
	e0ddr1-10-by-5-9	265	50	2	122	1	460	7%	8.420	4%	7.73	7.73	7.89	1,050.13	1,050.18	1,100.7	11.67	11.67	11.83
	e0ddr1-10-by-5-10	265	50	2	119	-	470	11%	8.540	6%	>7,200.00	>7,200.00	>7,200.00	>7,200.00	>7,200.00	>7,200.00	>7,200.00	>7,200.00	>7,200.00
	Total										>43,230.58	>37,858.63	>28,845.27	>47,398.8	>47,399.02	>28,850.98	>43,245.81	37,996.87	>21,848.25
jobShop-e0ddr2	e0ddr2-10-by-5-1	265	50	2	128	-	482	19%	8.840	3%	>7,200.00	>7,200.00	>7,200.00	>7,200.00	>7,200.00	>7,200.00	>7,200.00	>7,200.00	>7,200.00
	e0ddr2-10-by-5-2	265	50	2	127	-	469	21%	8.840	2%	>7,200.00	>7,200.00	>7,200.00	>7,200.00	>7,200.00	>7,200.00	>7,200.00	>7,200.00	>7,200.00
	e0ddr2-10-by-5-3	265	50	2	133	1	399	8%	7.960	11%	>7,200.00	>7,200.00	9.58	>7,200.00	>7,200.00	1,310.77	>7,200.00	>7,200.00	14.68
	e0ddr2-10-by-5-4	265	50	2	136	1	391	10%	7.960	9%	10.25	10.25	10.07	1,698.68	1,698.74	1,397.78	15.83	15.83	15.65
	e0ddr2-10-by-5-5	265	50	2	138	1	479	24%	9.100	3%	10.26	10.26	10.13	1,680.77	1,680.83	1,599.59	15.87	15.87	15.74
	e0ddr2-10-by-5-6	265	50	2	135	1	428	8%	8.240	3%	10.13	10.13	10.07	1,597.09	1,597.15	1,301.30	15.72	15.72	15.66
	e0ddr2-10-by-5-7	265	50	2	133	1	474	16%	8.800	4%	>7,200.00	>7,200.00	9.66	>7,200.00	>7,200.00	1,353.48	>7,200.00	>7,200.00	14.82
	e0ddr2-10-by-5-8	265	50	2	132	1	450	15%	8.540	2%	>7,200.00	>7,200.00	9.65	>7,200.00	>7,200.00	1,320.88	>7,200.00	>7,200.00	14.70
	e0ddr2-10-by-5-9	265	50	2	133	1	465	17%	8.800	4%	>7,200.00	>7,200.00	9.58	>7,200.00	>7,200.00	1,390.34	>7,200.00	>7,200.00	14.59
	e0ddr2-10-by-5-10	265	50	2	134	1	385	11%	7.960	7%	10.24	10.24	10.03	1,711.45	1,711.51	1,327.40	15.85	15.85	15.63
	Total										>43,240.88	>43,240.89	>14,478.76	>49,887.98	>49,888.22	>25,401.54	>43,263.27	43,263.27	>14,521.47
jobShop-enddr2	enddr2-10-by-5-1	265	50	2	139	1	482	19%	8.840	3%	>7,200.00	>7,200.00	>7,200.00	>7,200.00	>7,200.00	4,671.37	>7,200.00	>7,200.00	197.69
	enddr2-10-by-5-2	265	50	2	139	1	469	21%	8.840	2%	>7,200.00	>7,200.00	10.18	>7,200.00	>7,200.00	1,427.11	>7,200.00	>7,200.00	15.57
	enddr2-10-by-5-3	265	50	2	143	1	399	8%	7.960	11%	>7,200.00	>7,200.00	10.79	>7,200.00	>7,200.00	1,522.18	>7,200.00	>7,200.00	16.86
	enddr2-10-by-5-4	265	50	2	142	1	391	10%	7.960	9%	11.28	11.28	10.99	1,865.66	1,865.72	1,485.88	17.69	17.69	17.40
	enddr2-10-by-5-5	265	50	2	148	1	479	24%	9.100	3%	11.39	11.38	11.17	1,882.08	1,882.15	1,734.97	17.94	17.94	17.73
	enddr2-10-by-5-10	265	50	2	147	1	385	11%	7.960	7%	11.26	11.26	11.01	1,890.52	1,890.58	1,453.45	17.69	17.69	17.44
	Total										>21,633.93	>21,633.93	>7,254.14	>27,238.25	>27,238.45	12,294.97	>21,653.32	>21,653.33	282.68
pseudo-fpga	fpga-10-10	130	150	10	2	1	326	7%	6.173	6%	>7,200.00	4,124.71	0.20	>7,200.00	>7,200.00	0.47	>7,200.00	4,687.3	0.22
	fpga-10-8	106	120	10	2	1	233	8%	5.642	2%	>7,200.00	1,277.64	0.12	>7,200.00	2,316.88	0.25	>7,200.00	1,406.33	0.14
	fpga-10-9	118	135	10	2	1	280	7%	5.933	0%	>7,200.00	649.21	0.20	>7,200.00	845.65	0.36	>7,200.00	283.8	0.24
	fpga-11-10	141	165	11	2	1	353	7%	6.206	1%	>7,200.00	>7,200.00	0.27	>7,200.00	>7,200.00	0.65	>7,200.00	>7,200.00	0.31
	fpga-11-11	154	182	11	2	1	405	7%	6.412	3%	>7,200.00	>7,200.00	0.34	>7,200.00	>7,200.00	0.90	>7,200.00	>7,200.00	0.38
	fpga-11-9	128	149	11	2	1	301	7%	5.966	3%	>7,200.00	>7,200.00	0.37	>7,200.00	>7,200.00	0.50	>7,200.00	>7,200.00	0.44
	fpga-12-10	152	180	12	2	1	376	7%	6.233	1%	>7,200.00	>7,200.00	2.82	>7,200.00	>7,200.00	1.02	>7,200.00	>7,200.00	1.57
	fpga-12-11	166	198	12	2	1	435	6%	6.429	1%	>7,200.00	>7,200.00	461.60	>7,200.00	>7,200.00	5.71	>7,200.00	>7,200.00	4.59
	fpga-12-12	180	216	12	2	1	492	7%	6.63	0%	>7,200.00	>7,200.00	0.56	>7,200.00	>7,200.00	1.60	>7,200.00	>7,200.00	0.63
	fpga-13-11	178	215	13	2	1	461	7%	6.502	1%	>7,200.00	>7,200.00	0.66	>7,200.00	>7,200.00	1.82	>7,200.00	>7,200.00	0.76
	fpga-13-12	193	234	13	2	1	525	6%	6.679	1%	>7,200.00	>7,200.00	0.78	>7,200.00	>7,200.00	2.34	>7,200.00	>7,200.00	0.90
	fpga-13-13	208	254	13	2	1	588	7%	6.878	1%	>7,200.00	>7,200.00	0.94	>7,200.00	>7,200.00	3.07	>7,200.00	>7,200.00	1.08
	fpga-14-12	206	252	14	2	1	553	7%	6.722	0%	>7,200.00	>7,200.00	14.21	>7,200.00	>7,200.00	14.24	>7,200.00	>7,200.00	14.55
	fpga-14-13	222	273	14	2	1	624	6%	6.956	1%	>7,200.00	>7,200.00	1.38	>7,200.00	>7,200.00	4.59	>7,200.00	>7,200.00	1.63
	fpga-14-14	238	294	14	2	1	692	7%	7.122	0%	>7,200.00	>7,200.00	1.62	>7,200.00	>7,200.00	5.83	>7,200.00	>7,200.00	1.89
	fpga-15-13	236	293	15	2	1	655	7%	6.939	0%	>7,200.00	>7,200.00	2.17	>7,200.00	>7,200.00	7.58	>7,200.00	>7,200.00	2.64
	fpga-15-14	253	315	15	2	1	731	6%	7.108	1%	>7,200.00	>7,200.00	2.48	>7,200.00	>7,200.00	9.30	>7,200.00	>7,200.00	2.99
	fpga-15-15	270	338	15	2	1	805	7%	7.263	1%	>7,200.00	>7,200.00	2.84	>7,200.00	>7,200.00	11.78	>7,200.00	>7,200.00	3.41
	fpga-20-18	416	540	20	2	-	1288	7%	7.852	2%	>7,200.00	>7,200.00	>7,200.00	>7,200.00	>7,200.00	>7,200.00	>7,200.00	>7,200.00	>7,200.00
	fpga-20-19	438	570	20	2	1	1395	6%	8.032	0%	>7,200.00	>7,200.00	83.64	>7,200.00	>7,200.00	449.32	>7,200.00	>7,200.00	109.39
	fpga-20-20	460	600	20	2	1	1496	6%	8.093	3%	>7,200.00	>7,200.00	74.45	>7,200.00	>7,200.00	426.32	>7,200.00	>7,200.00	123.63
	Total										>151,200.00	>135,651.56	>7,851.66	>151,200.00	>13,9962.53	>8,147.67	>151,200.00	>135,977.42	>7,471.40