

Using Weisfeiler-Leman Features for Algorithm Selection in Constraint Optimisation

Alessio Pellegrino, Jacopo Mauro
University of Southern Denmark, Denmark
{alessio, mauro}@imada.sdu.dk

Abstract

Algorithm Selection is essential for efficient Constraint Programming. Over the years, many algorithm selectors based on machine learning methods have been successfully applied, yet traditional feature extraction methods often rely on manually decided instance-level statistics that fail to capture the underlying problem structure.

In this paper we aim to bridge this gap by introducing a novel, automated feature extraction methodology that integrates graph conversion and Weisfeiler-Lehman graph kernels to generate robust structural representations of problem instances. The 1-WL test bounds the graph-distinguishing power of standard message-passing Graph Neural Networks (GNNs), and suitable GNN architectures match this bound [Xu *et al.*, 2018]. WL-based features offer an alternative that does not require training a GNN. Our primary contribution is a cut-based representation (WLC) designed to model structural partitions and provide a more nuanced predictive signal. We evaluate our approach on instances from the 2023–2025 MiniZinc Challenges across two tasks: maximizing Borda count scores and maximizing predictive accuracy. Experimental results across Support Vector Machines, Random Forests, and Multi-Layer Perceptrons demonstrate that cut-based features outperform `fzn2feat` with SVMs, while results with RFs and MLPs are closer.

1 Introduction

Algorithm Selection, the process of selecting the most appropriate algorithm from a portfolio for a given problem instance, is a critical component in the efficient execution of Constraint Programming models, as solver performance varies significantly across different problem instances. Traditional feature extraction methodologies, such as `fzn2feat`, primarily rely on instance-level statistics that may fail to capture the underlying latent structure or the complex interconnections between variables. While graph-based representations are designed to capture these structural nuances, they are typically coupled with deep neural networks that require

substantial computational overhead. This creates a representational gap: non-neural machine learning techniques remain largely “structure-blind,” while structure-aware models are often too expensive for practical deployment.

This paper proposes a novel methodology to bridge this gap by implementing a graph conversion algorithm compatible with both classical and deep machine learning paradigms. Our approach utilizes Weisfeiler-Lehman graph kernels to generate robust representations of problem instances [Morris *et al.*, 2023]. The primary innovation is the introduction of a cut-based representation, which explicitly models structural partitions to provide a more nuanced predictive signal than standard graph features.

To evaluate the efficacy of these features, we constructed two distinct algorithm selection tasks using instances from the 2023, 2024, and 2025 MiniZinc Challenges.¹ Experimental results across Support Vector Machines, Random Forests, and Multi-Layer Perceptrons demonstrate that our cut-based features outperform `fzn2feat` on both tasks when paired with SVMs. Comparisons with RFs and MLPs show smaller differences, with `fzn2feat` achieving slightly better results in several RF configurations. Our work effectively demonstrates that graph-based structural information can be leveraged to enhance traditional machine learning architectures, ensuring performance gains that are robust across diverse algorithmic structures.

2 Background

This section provides the theoretical foundation for this work.

2.1 Combinatorial Optimisation

Constraint Programming (CP) [Rossi *et al.*, 2006] is a paradigm for solving combinatorial optimisation problems (COPs) by defining them via variables, domains, and constraints. CP utilises global constraints (such as *all-different*, *cumulative*, or *circuit*) which capture complex substructures. These constraints use specialised filtering algorithms to prune the search space through propagation, enabling more efficient modelling and solving.

Constraint solvers navigate the solution space using techniques like constraint propagation [Bessiere, 2006], branch and bound [Boyd and Mattingley, 2007], and conflict-driven

¹<https://www.minizinc.org/challenge/>

clause learning (CDCL) [Marques-Silva *et al.*, 2009]. Given that COPs are often NP-hard [Papadimitriou, 1994], parallelisation is frequently employed to speed up the computation. This typically follows two patterns: (i) the job-splitting approach, which parallelises a single solver’s strategy, and (ii) the portfolio approach, which runs multiple solvers with different strategies simultaneously [Amadini *et al.*, 2016]. Comprehensive reviews for parallelising these specific techniques are available in the literature [Gent *et al.*, 2018; Crainic *et al.*, 2006; Martins *et al.*, 2012]. Surveys of algorithm selection are provided by Kerschke *et al.* [2019]; Kotthoff [2016].

2.2 The Algorithm Selection Problem

Algorithm Selection (AS) involves identifying the most suitable algorithm for a specific problem instance to optimise metrics like computational efficiency or accuracy. This process is underpinned by the No Free Lunch theorem [Wolpert and Macready, 1997], which, under its assumptions, rules out an algorithm that outperforms all others across all objective functions. This motivates selecting algorithms according to the problem context, since solver performance can vary across instances.

Formally, given a problem instance $x \in \mathcal{P}$ drawn from an instance space $\mathcal{P}$ and a portfolio of candidate algorithms $\mathcal{A}$, the objective of an algorithm selector is to determine a mapping $S : \mathcal{P} \rightarrow \mathcal{A}$ that optimizes an expected performance metric $m : \mathcal{A} \times \mathcal{P} \rightarrow \mathbb{R}$ (such as runtime, penalized average runtime, or solution quality). In practice, machine learning models parameterise this mapping by first transforming the instance x into an informative feature vector $\mathbf{f}(x) \in \mathbb{R}^d$.

2.3 The WL Algorithm

The original purpose of the WL algorithm is to check if two graphs are isomorphic [Kiefer, 2020]. It does so by assigning a colour to each node on a graph and then iteratively refining it by aggregating its colour with the colours of the neighbouring nodes. The algorithm keeps refining the colours until either (a) the two graphs have different colour histograms (meaning they are non-isomorphic) or (b) the refinement does not change the induced partitioning of the graphs (meaning that the graphs are potentially isomorphic).

It is important to note that the WL algorithm is not a complete isomorphism test: two non-isomorphic graphs can remain indistinguishable. For example, equally sized regular graphs of the same degree remain indistinguishable by 1-WL when all nodes initially have the same colour.

It is also possible to generalise the WL algorithm to the k -WL algorithm, where the colours are assigned to a k -tuple of nodes. In this sense, the standard WL algorithm is a special case of the k -WL algorithm where $k = 1$. As k grows, the k -WL algorithm becomes much more powerful in distinguishing non-isomorphic graphs; however, its computational complexity also grows exponentially in k . Thus, for the purpose of this paper, we only focus on the standard 1-WL algorithm.

3 Related Work

In this section, we review related literature on algorithm selection methods and their feature representations, graph rep-

resentations of constraint problems, and graph feature extraction techniques.

3.1 Algorithm Selectors and Handcrafted Features

Most modern algorithm selectors employ machine learning techniques to predict which algorithm will perform best on an unseen problem instance. In the SAT domain, SATzilla [Xu *et al.*, 2012] uses Random Forests over portfolios of SAT solvers. In CP, approaches like CPHydra [Bridge *et al.*, 2012] and SUNNY [Amadini *et al.*, 2014a] utilise k -nearest neighbours to generate instance-specific solver schedules.

A central challenge in algorithm selection is constructing informative feature representations of problem instances. Traditionally, selectors have relied on manually designed, domain-specific features. For instance, SUNNY relies on `fzn2feat` [Amadini *et al.*, 2014b], which extracts static features from a FlatZinc model, such as the counts and types of variables and constraints, domain sizes, and basic metrics derived from the variable-constraint graph (e.g., degree statistics). The `mzn2feat` 1.2.1 tool, which includes the FlatZinc extractor `fzn2feat`, provides 95 features.² However, such handcrafted features are expensive to design, often fail to generalize across distinct problem domains, and compress complex combinatorial substructures into flat summary statistics.

3.2 Graph Representations for Constraint Problems

While representing combinatorial problems as graphs is a conceptually intuitive approach [Bockmayr and Hooker, 2005], determining the optimal structural configuration for such graphs remains a non-trivial challenge. Over the years, many different approaches have emerged: a lot of MIP-related tasks employ a bipartite graph representation where a pair of variable/constraint nodes share an edge when the variable has a non-zero coefficient in the constraint [Khalil *et al.*, 2022]. Another approach, often used in CP, is to represent the problem as a tripartite graph, where nodes are constraints, variables and values in the variables’ domains. Variable and constraint nodes share edges if the variable participates in the constraint. Furthermore, variable nodes also have edges that connect them to all possible values in their domain [Cappart *et al.*, 2023]. Finally, Boisvert *et al.* [2024] decide to decompose constraints into smaller components, and the final graph is akin to the abstract syntax tree of a program.

3.3 Graph Kernels and GNNs in Combinatorial Optimisation

Feature extraction transforms graph-structured data into fixed-dimensional vector spaces [Borgwardt *et al.*, 2020]. Classic methods rely on graph-theoretical metrics to describe structural properties. Node-level features describe a node’s topological importance, including Degree Centrality, Clustering Coefficient [Watts and Strogatz, 1998], and Betweenness or Closeness Centrality [Newman, 2010]. Edge-level features are primarily used for link prediction and node proximity,

²<https://github.com/CP-Unibo/mzn2feat>

such as Common Neighbours, Jaccard Coefficient, Adamic-Adar Index [Adamic and Adar, 2003], and Katz Index [Katz, 1953].

Graph-level features are often extracted via graph kernels that compute similarity as a scalar product between graphs:

- **Weisfeiler-Lehman (WL) Kernel** [Shervashidze *et al.*, 2011]: Iteratively relabels nodes based on neighbor signatures (see Section 5.1 for more details on the extraction process).
- **Graphlet Kernel** [Borgwardt *et al.*, 2020]: Counts small, non-isomorphic subgraphs.
- **Random Walk Kernel** [Vishwanathan *et al.*, 2010]: Compares graphs based on matching paths.

WL-based features are used in cheminformatics [Young, 2022] as well as in neighbouring fields like automated planning [Chen *et al.*, 2024]. Graph neural networks have also been applied to computer vision, for example to point cloud classification [Simonovsky and Komodakis, 2017].

In recent years, Graph Neural Networks (GNNs) have gained significant popularity for combinatorial optimisation tasks. Crucially, the message-passing mechanism in standard GNNs is fundamentally connected to and bounded in expressive power by the 1-WL test [Xu *et al.*, 2018; Morris *et al.*, 2023]. However, while GNNs learn continuous representations via deep architectures, they require substantial training data and incur non-trivial inference latency when querying instances. For algorithm selection, where feature extraction and selector inference must be lightweight to avoid offsetting the performance gains of the selected solver, graph kernels based on Weisfeiler-Leman refinement provide a compelling alternative that preserves theoretical expressiveness while remaining fast to compute.

4 Graph Conversion

In this section we describe the graph conversion we propose in this work. Since the graph conversion is decoupled from the feature extraction methodology, we will separate the two sections and defer the feature extraction methodology discussion to Section 5. As stated in Section 3.2, many different approaches for graph conversions have been proposed. Our approach is inspired by that of Boisvert *et al.* [2024]. We have decided to use Boisvert *et al.*’s syntax tree-like representation as it allows us to decompose complex constraints into simpler elements that could be reused in different constraints with similar meanings.

Unlike Boisvert *et al.*, we do not include domain values in our representation. We decided to exclude domain values for practical reasons. They argue that a good conversion should be injective (e.g. two distinct instances should never be mapped to the same graph). We do recognise the soundness of this argument, however, our primary objective is to use the features for algorithm selection and, as an example, an instance from the 2025 Minizinc challenge³ (coming from the cgt problem with instance name `cgt_12_h27_r0.05_s0.5_3`) had an objective function with a domain between -1.932.060

and 1.714.644, resulting in 3.646.705 total possible values just for one domain. The full instance would also require adding other variables and their domains as well as constraints and operators. A graph with this many nodes would be intractable for our purposes. Furthermore, even assuming the tractability of its conversion, the high number of value nodes would add a lot of noise that could weaken the signal from the remaining nodes. Finally, our graphs are directed instead of undirected. This choice gives shallow local constraint decompositions, making it possible to propagate information along their directed paths with a limited number of aggregation steps.

In more detail, we define five types of nodes:

- **Variable nodes**: representing the variables of the problem.
- **Solution node**: a single node for each problem that represents the type of problem (Minimisation, Maximisation or Satisfaction).
- **Parameter/Literal nodes**: representing the literal values of the problem (e.g. 1, 2, True, False, etc.).
- **Operator nodes**: representing arithmetic and functional operations between variables, parameters and other operators (e.g. +, ×, ÷ etc.) Operator nodes have a sub-category for the type of operator represented.
- **Constraint nodes**: representing relational predicates (e.g. =, ≠, <, ≤), logical operations (e.g. ∧, ∨, ¬), and global constraints (e.g. `all_different`, `table`). Constraint nodes also have a sub-category for the type of constraint the node represents.

All node and sub-node types we have defined for the conversion from FlatZinc can be seen in the Appendix A.

As previously mentioned, edges between nodes are directed: variables and parameter nodes have no incoming nodes and only point to the operators or constraints they take part in. There is only one exception: for optimisation problems, the variable containing the objective value to optimise has an incoming edge from the Solution node. In case of Satisfaction problems, the solution node is isolated. Operator nodes can have both incoming and outgoing edges. Incoming edges come from variables, parameters and other operators that take part in the operation. Outgoing edges go to other operator nodes or to constraint nodes. Constraint nodes can also have edges to other constraint nodes when they appear as a sub-expression to other constraints. As an example, the expression $(x = 3) \wedge (y \neq 5)$, will have three constraint nodes ($=$, $\wedge$ and $\neq$) with $=$ and $\neq$ having exiting edges to $\wedge$.

Edges are typed as well; in particular, there are three types of edges:

- **Lead edges**: used to connect variables, parameters and operators to operators and constraints for which the order of the operands does not matter (e.g. sum or equality). Lead Edges are also used for variables, parameters and operators that fall on the left side of an operation or constraint where the order of the operands matters (e.g. subtraction or strictly less than). Lead edges are also used to connect the solution node to the objective variable in optimisation problems.

³<https://www.minizinc.org/challenge/2025/results/>

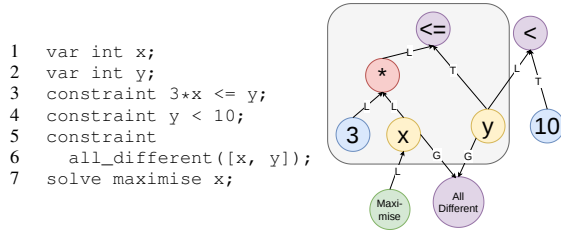


Figure 1: Conversion of a constraint model (left) to a graph (right). Variables in yellow, Parameters in blue, Operators in red, Constraints in purple, and Solve in green. The area in gray represents the constraint $3 \times x \leq y$. Lead edges are labelled as L, Trailing as T and Global as G.

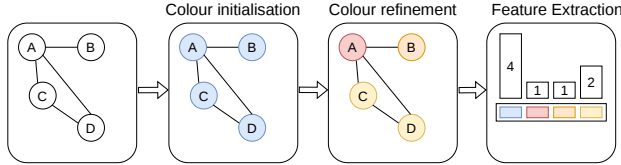


Figure 2: Feature extraction process using the WL algorithm on a simple, undirected graph. The colour refinement is executed only once. At first, all nodes get the same colour, then node A aggregates its colour (blue) with its 3 neighbours’ colours (blue), getting, as a result, (blue, blue,blue,blue). This produces a new colour, red. C and D, having only two neighbours, get, as aggregation (blue, blue,blue), resulting in the new colour yellow and, finally, node B, having only one neighbour, aggregates to (blue, blue), getting as new colour orange. In the final steps, we count the number of occurrences of each colour, and we get 4 blues from the original initialisation and 1 red, 1 orange and 2 yellow after the first aggregation step.

- **Trailing edges:** Used to connect variables, parameters and operators to operators and constraints for which the order of the operands matters and the operands fall on the right side of the operation.
- **Global edges:** Used to connect variables and parameters to global constraints. This helps to distinguish connections to ordinary constraints from connections to global constraints.

Figure 1 shows an example of how a simple constraint model can be converted using our conversion methodology.

5 Extracting Features Using The Weisfeiler-Leman Algorithm

Once we have defined how to convert any instance to a graph, we can use the WL algorithm to extract features from the graphs.

5.1 WL For Feature Extraction

To use the WL algorithm for feature extraction, we repeat the colour refinement process for a pre-defined number of steps on a single graph, then we extract features by creating a histogram by counting the colors. Then, the final feature vector is created by concatenating the histograms after each iteration. A depiction of the execution of the WL algorithm for

feature extraction (with one aggregation step) can be seen in Figure 2.

Note that while the standard WL feature extraction process typically concatenates histograms from all iterations, we deliberately utilise only the final histogram. This decision is informed by several theoretical and practical considerations specific to our domain.

First, from a theoretical perspective, the final iteration’s labels represent the most refined partition of the graph’s nodes. As demonstrated in the analysis of Graph Isomorphism Networks [Xu *et al.*, 2018], the final state of a message-passing scheme (given an injective aggregation function) effectively encodes the full subtree structure. In our discrete WL implementation, the final labels serve as a compressed representation that implicitly preserves the structural history of previous iterations.

Furthermore, the specific topology of our data justifies this approach. The extracted graphs are characterised by a shallow diameter and directed edges, which significantly limit the depth of information flow. In such instances, the WL process converges rapidly as nodes quickly incorporate all available predecessors’ information. Our empirical analysis (in Appendix F) across two distinct tasks and three different ML models confirmed this: we found no statistically significant difference in performance between concatenated features and the single-histogram approach except for one SVM accuracy comparison, which favoured the single-histogram approach. Consequently, we prioritise the latter.

5.2 Enhancing Features’ Expressiveness

The simplest WL feature variant starts with identical node colours and does not use edge types. However, in our case, we have both node and edge-level information we can use to better represent the input graph. For this reason, inspired by previous work of Togninalli *et al.* [2019], we include two extensions to the base algorithm:

- **Node-information extension:** To add node information into the extraction process, we initialise the node colour with the hash of its type and subtype (e.g. a `Variable` node and a `Sum` node will be initialised with different colours). Formally, $c^{(0)}(v) \leftarrow \text{HASH}(\text{type}(v) \parallel \text{subtype}(v))$ where $\parallel$ denotes string concatenation.
- **Edge-information extension:** To add edge information into the extraction process, we concatenate the edge type to the node colour during the aggregation step (e.g. in the example of Figure 2, if node A pointed to node B via a Lead edge, the aggregation of node B would be: (blue, blue-L)). Formally, assuming a directed graph where edges flow towards v , its multiset update becomes $M(v) \leftarrow \{c^{(t-1)}(w) \parallel \text{type}(w, v) \mid (w, v) \in E\}$.

Finally, since our graphs are directed, we can filter out outgoing edges and update the color of a node solely based on the incoming edges.

Using these extensions, we can now define four types of features: (i) `WL`, standard WL features without node or edge information, (ii) `WLn`, WL features enhanced via node-information extensions, (iii) `WLe`, WL features enhanced via

edge-information extensions, and (iv) WL_{ne} , WL features enhanced with both edge and node-information.

5.3 The “Cut”

In the WL algorithm, the colour of a node is highly dependent on its degree, and two nodes with different degrees, even when sharing the same original colour and the same type of neighbour nodes, will end up with different colours. This can be problematic for nodes representing operators and constraints where the number of connected sub-elements can vary. As an example, let’s examine the `all_different` global constraint. Using the standard algorithm, `all_different([v1, v2, v3])` will get a totally different and uncorrelated colour to `all_different([v1, v2])` solely due to its degree. In practice, while preserving some differences between the two constraints, we would still capture the fact that they are similar.

To achieve this, we can virtually “cut” the connections between the nodes representing constraints and operators with a variable number of operands in them. We will refer to these nodes as cut nodes. In practice, during the aggregation steps, the cut nodes will not be updated since their incoming connections will be ignored. However, the cut nodes will still contribute to the update of the nodes they are connected to via their outgoing edges. All cut nodes can be seen in Appendix A.

After the aggregation steps have been completed, each cut node’s colour can be updated by aggregating it with the set (not multiset) comprised of all its connected, original node colours. By using a set rather than a multiset, cut nodes with the same initial colour and the same set of incoming node types will receive the same colour, independent of their in-degree.

The cut shifts the local equivalence criterion from a graded/counting bisimilarity (which tracks exact quantities via multisets) closer to a standard bisimilarity (which tracks only the existence of neighbor types via sets).

Since a cut removes any information regarding the degree of a node, to retain it we add extra features after the colour histogram. Specifically, from each cut node, we can generate node pairs comprising the cut node and its incoming neighbours. Thus, each cut node of in-degree n will generate n pairs. After having generated all the pairs, we count the occurrence of each pair in a graph to generate a histogram of pairs and concatenate it to the histogram of colours.

The cut can only be applied when node information is present, thus, only to WL_n and WL_{ne} features. We will call the features where the cut is applied WLC_n if only the cut is applied to WL_n features and WLC_{ne} if the cut is applied to WL_{ne} . A full formalisation and line-by-line description of the WLC_n feature extraction procedure is detailed in Algorithm 1 in Appendix B, while a graphical depiction of how the cut is applied can be seen in Figure 3.

5.4 Auxiliary Structural Features

As a consequence of our design choices, especially having directed graphs, variable and parameter nodes have no incoming edges (with the sole exception of the objective variable

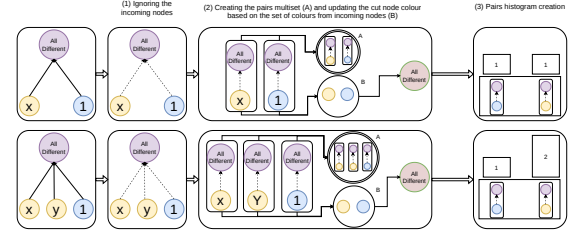


Figure 3: Application of the cut on two occurrences of the `all_different` constraint (top and bottom). Constraint nodes are in purple, variable nodes in yellow and parameter nodes in blue. First, cut nodes do not perform the normal WL aggregation steps (1), then we generate all pairs and update the colour of the `all_different` node by aggregating it with the set of all types of nodes with incoming edges (2) and then we generate the pairs histogram by counting all the occurrences of the pairs (3). In this example, since both constraints share the same type of incoming nodes, they will be coloured in the same way. However, having a different node degree will result in a different pair histogram.

connected to the Solution node). Nodes of the same source type remain identically coloured within each refinement iteration, so their histogram entries reflect their counts. The objective variable is an exception because it has an incoming edge.

To provide richer structural context beyond raw counts, we augment the histogram representation with two auxiliary features: (i) the average number of constraints and operators in which a variable participates (variable out-degree), and (ii) the average number of operations in which a parameter participates (parameter out-degree).

5.5 Feature Normalisation and Vocabulary Alignment

Once the raw feature histograms are extracted, two post-processing steps are applied to produce a standardized, model-ready feature representation:

Normalisation. Raw histogram frequencies scale proportionally with problem instance size. Without normalisation, feature magnitudes reflect instance scale rather than topological structure, hindering cross-instance generalisation. To ensure comparability across diverse problems, we normalise the colour histogram by the total number of nodes $|V|$ and the pairs histogram by $\max(1, |Pairs|)$. This bounds all histogram values within $[0, 1]$.

Vocabulary Alignment. Across distinct problem instances, the set of generated colours varies, with different graphs sharing only a subset of labels. To construct a consistent vector representation for machine learning models, we assemble a global colour vocabulary across all training graphs. Any colour missing from a particular instance is assigned a frequency of zero. During inference on unseen test instances, any novel colours not observed in the training vocabulary are discarded.

5.6 Remarks on Graph Dynamics and Depth

Due to the directed nature of our converted graphs, paths from sources to sinks are inherently shallow, especially after apply-

ing the cut. For example, in a constraint such as $x+3 = y$, assuming x and y are not objective variables, information from the source nodes reaches the operator node (+) after one iteration and the equality constraint node (=) after two. The hash labels can still change in subsequent iterations because each update includes the previous label.

The local constraint decompositions have short directed paths (typically between 1 and 4 edges, with effective propagation depth reduced to 2 under the cut in the example in Appendix C). This local depth should not be confused with the diameter of the entire instance graph. The final set-based update of cut nodes deliberately ignores incoming multiplicities; it therefore differs from standard multiset-based WL refinement.

6 Experimental Results

To evaluate the proposed methodology, we implemented the graph conversion algorithm within the FlatZinc modelling ecosystem. We used the 300 instances from the 2023, 2024, and 2025 MiniZinc Challenges to construct two algorithm selection tasks: (i) maximising the Borda count score across three solvers, (ii) maximising prediction accuracy for the same solver set. Solvers were executed with the `--free-search` flag enabled⁴ on an Intel Xeon Gold 6130 processor with a 20-minute timeout.

Our approach is compared against two baselines: a standard performance baseline (single best solver (SBS) or majority classifier) and `fzn2feat`.

Given the limited depth of our graph representations (see Section 5.6), we evaluate WL features using one and two aggregation steps. The number of aggregation steps is appended to the features using a dash (-) sign (example: `WLc-n` features with one aggregation will be `WLc-n-1`).

For each task, we trained three models: Support Vector Machines (SVM), Random Forests (RF), and Multi-Layer Perceptrons (MLP) with three hidden layers. The selection of SVM, RF, and MLP provides an evaluation framework that tests feature sets against three distinct mathematical paradigms: geometric margin optimisation, ensemble-based logical partitioning, and connectionist functional approximation. All tasks have been evaluated using repeated five-fold nested cross-validation on 10 random seeds for a total of 50 outer-fold evaluations per task per ML model. The training pipeline followed a four-step protocol:

1. **Hyperparameter Optimisation :** We performed a grid search over predefined configurations, evaluating each via mean 5-fold internal cross-validation.
2. **Model Selection:** We selected the highest-scoring configuration. Ties were broken by prioritising simpler models to mitigate overfitting:
 - For SVM: lower C (stronger regularisation) and γ values, favoring the RBF kernel.
 - For RF: fewer trees and shallower maximum depth.
 - For MLP: fewer training steps and a smaller α penalty.

⁴https://docs.minizinc.dev/en/stable/command_line.html#cmdoption-f

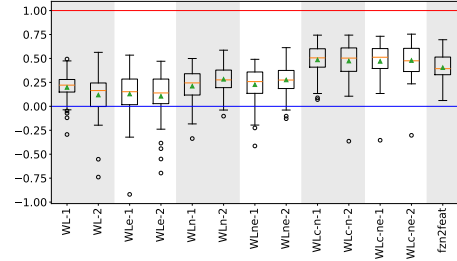


Figure 4: Performance results for SVM models trained on WL-based and `fzn2feat` features for the Borda count maximization task. Scores (top) are normalized as $S_{norm} = \frac{S - S_{sbs}}{S_{vbs} - S_{sbs}}$, where S is the model score, sbs is the Single Best Solver, and vbs is the Virtual Best Solver.

3. **Dimensionality Reduction:** We evaluated the best configuration on reduced feature sets using Principal Component Analysis (PCA) in the range from 20 to $\min(n_features, n_samples)$ with a step size of 20.

4. **Final Feature Selection:** The feature set (original or PCA-transformed) yielding the highest cross-validation score was selected for final evaluation.

The last two steps (Dimensionality Reduction and Final Feature Selection) have been skipped for MLP classifiers due to their ability to handle high-dimensional data.

All figures group together features extracted with the same methodology but with different aggregation steps, using an alternating shaded background to improve readability.

6.1 Borda Count Score Maximization

In this task, we compare three sequential solvers: `OR-Tools` (CP-SAT) [Google OR-Tools], `Chuffed` [Chu *et al.*], and `CPLEX` [Nickel *et al.*, 2022] on the 300 instances. Scores were assigned based on the Borda count strategy used in the MiniZinc Challenge.⁵ We framed this as a classification problem where the target label is the solver achieving the highest score. The results and Borda count scores reflect only the solver execution time, excluding feature extraction costs.

We excluded one trivially unsatisfiable instance from which features could not be extracted, as well as instances where no solver reached a solution. The final dataset comprised 289 instances. We employed `StratifiedKfold` from `Scikit-learn`⁶, stratifying based on the performance gap between the Virtual Best Solver (VBS) and the worst-performing solver in the portfolio.

Figure 4 illustrates the performance of SVM models trained on WL-based features. While all feature sets generally outperformed the baseline, incorporating node information (`WLn`) yielded a significant performance increase over standard WL features. Conversely, edge information (`WLc`) provided no substantial predictive signal. Combining node and edge attributes (`WLne`) did not further improve results beyond the node-only baseline.

⁵<https://www.minizinc.org/challenge/2025/>

⁶https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.StratifiedKfold.html

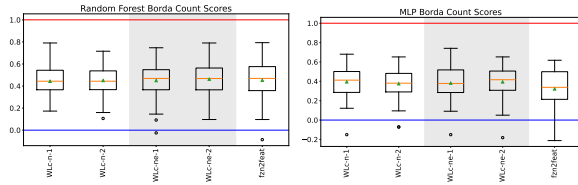


Figure 5: Comparative performance of cut-based features ($WLC-n/WLC-ne$) against the `fzn2feat` baseline for the Borda count maximization task. Scores are normalised relative to the Single Best Solver and Virtual Best Solver. Results on the left are obtained by training RF models, while those on the right are obtained by training MLPs.

	Mean	Median
<code>fzn2feat</code>	0.45	0.47
$WLC-n-1$	0.48	0.51
$WLC-n-2$	0.47	0.50
$WLC-ne-1$	0.46	0.51
$WLC-ne-2$	0.48	0.48

Table 1: Mean and median results of cut features and `fzn2feat`. Results show the best performing scores obtained using SVM for cut features and RF for `fzn2feat`.

Notably, features derived from the cut-based representation ($WLC-n$ and $WLC-ne$) significantly outperformed all other variants. The number of aggregation steps had a negligible impact on final performance across all feature sets.

MLP classifiers exhibit performance trends similar to those of the SVM on WL -based features. On the other hand, RF ones show an increase when adding the node information, but not a significant jump between $WLn/WLne$ features and the cut ones and these feature sets have similar performance. For readability purposes and since cut features are either on par or better than their non-cut counterpart, we will now focus only on the performance of cut-based features relative to the `fzn2feat` baseline for these models. We will provide their comprehensive comparison in Appendix D.

Figure 5 illustrates the comparative performance of RF and MLP models. For the RF, `fzn2feat` achieves marginally higher scores than the cut-based features, with the exception of $WLC-ne-2$, which still presents slightly higher mean and median scores.

Finally, we conducted a cross-model comparison between the best-performing `fzn2feat` configuration and the best cut-based configuration (SVM) (mean). The results are summarised in Table 1.

6.2 Algorithm Selection with Accuracy Maximisation

The setup for this task was the same as for the Borda count maximisation one (in Section 6.1), with the only difference being the objective of the task, which changes from maximising the score of the portfolio to maximising predictive accuracy. In this task, the baseline to beat is the majority classifier, which always predicts the most frequent label.

The experimental configuration for this task mirrors the

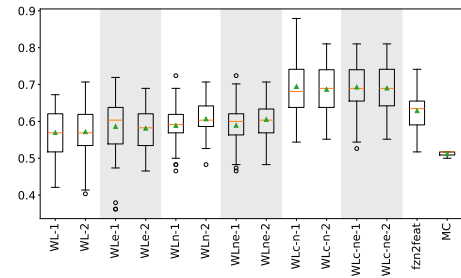


Figure 6: Predictive accuracy for SVM models trained on WL -based and `fzn2feat` features.

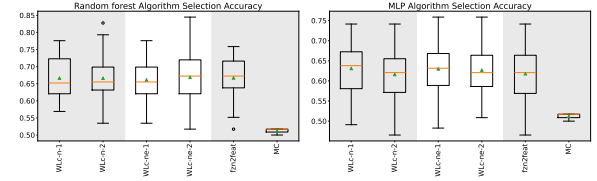


Figure 7: Comparative performance of cut-based features ($WLC-n/WLC-ne$) against `fzn2feat` and the majority classifier (MC) baseline. Results on the left are obtained by training RF models, while those on the right are obtained by training MLPs.

Borda count maximisation setup (Section 6.1), with the objective shifted to maximising predictive accuracy. This change of objective was also reflected in the stratification strategy, which is based on the final label instead of the gap.

Consistent with the observations in Section 6.1, the comparisons of WL -based features for SVM models (Figure 6) indicate that cut-based features significantly outperform alternative variants. However, for the accuracy maximisation task, the marginal benefit of incorporating node information is less pronounced than in the Borda count task. Regardless, all feature configurations demonstrate a significant improvement over the majority classifier baseline.

When compared against the `fzn2feat` baseline (Figure 6), cut-based features exhibit significantly higher accuracy.

As in Section 6.1, RF models show similar performance with cut and non-cut features that include node information. However, MLP-based models demonstrate a significant improvement when paired with cut features. We show a comprehensive comparison in Appendix E, while here we show only cut-based features comparisons with `fzn2feat` when paired with RF and MLP models.

Figure 7 shows the performance of RF and MLP-based classifiers trained with cut-based and `fzn2feat` features. While `fzn2feat` features perform slightly better on the RF classifier, the differences are minimal. Similarly small differences can be seen for the MLP-based classifiers, where all cut-based features perform slightly better than `fzn2feat`.

Table 2 pairs the best models for each feature type. Again, cut features obtain better mean and median results than `fzn2feat`.

	Mean	Median
fzn2feat	0.66	0.67
Wlc-n-1	0.69	0.68
Wlc-n-2	0.69	0.69
Wlc-ne-1	0.69	0.69
Wlc-ne-2	0.69	0.69

Table 2: Mean and median results of cut features and `fzn2feat`. Results show the best performing scores obtained using SVM for cut features and RF for `fzn2feat`.

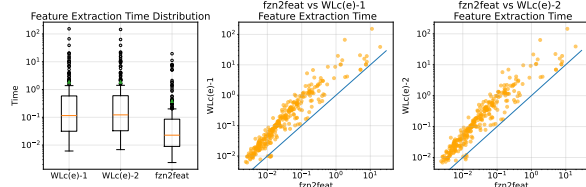


Figure 8: Feature extraction cost comparison between WLC-n/WLC-ne and `fzn2feat`. The distribution of extraction times (left) is supplemented by pairwise performance comparisons (centre, right). The blue line corresponds to $x = y$.

6.3 Feature Extraction Cost

The computational overhead for extracting cut-based features compared to `fzn2feat` is illustrated in Figure 8. Because both methodologies necessitate a preliminary compilation from MiniZinc to FlatZinc, we exclude compilation time from the reported extraction costs to isolate the performance of the feature generators themselves.

While `fzn2feat` exhibits lower absolute extraction times, the vast majority of instances require less than one second, even when utilising cut-based features. It is important to note that the current implementation of the WL feature extraction pipeline (comprising graph conversion and Weisfeiler-Lehman aggregation) is not fully optimised. Significant performance gains could be realised through algorithmic refinements and the parallelisation of independent graph operations, further reducing the overhead for large-scale instance sets.

7 Conclusions and Discussion

This paper introduced a graph-based feature extraction methodology for Constraint Programming instances, with a specific focus on cut-based representations (WLC). By integrating this process directly into the FlatZinc ecosystem, we bridged the gap between structural graph representations and classical machine learning paradigms.

Our findings lead to several key conclusions regarding the efficacy of these features: (i) **Robustness Across Paradigms:** The WL and WLC features demonstrate high versatility, providing strong predictive signals across different machine learning models. (ii) **Stability and Performance Reliability:** The proposed features exhibit high stability across diverse tasks.

Future work will focus on optimising the computational efficiency of the extraction pipeline and exploring the inte-

gration of these features into dynamic, online algorithm selection frameworks. Furthermore, we will explore the effect of combining our feature sets with classical feature sets (like `fzn2feat`) to evaluate if combining different types of signals can improve the predictive performance. We will also focus on generalising our features to different tasks not strictly related to algorithm selection but still relevant in the CP community.

References

- Lada A Adamic and Eytan Adar. Friends and neighbors on the web. *Social networks*, 25(3):211–230, 2003.
- R. Amadini, M. Gabbrielli, and J. Mauro. SUNNY: a lazy portfolio approach for constraint solving. *Theory and Practice of Logic Programming*, 14(4-5):509–524, 2014.
- Roberto Amadini, Maurizio Gabbrielli, and Jacopo Mauro. An enhanced features extractor for a portfolio of constraint solvers. In *Proceedings of the 29th annual ACM symposium on applied computing*, pages 1357–1359, 2014.
- Roberto Amadini, Maurizio Gabbrielli, and Jacopo Mauro. Portfolio approaches for constraint optimization problems. *Annals of Mathematics and Artificial Intelligence*, 76(1-2):229–246, 2016.
- Christian Bessiere. Constraint propagation. In *Handbook of Constraint Programming*, volume 2 of *Foundations of Artificial Intelligence*, pages 29–83. Elsevier, 2006.
- Alexander Bockmayr and John N Hooker. Constraint programming. In *Discrete Optimization*, volume 12 of *Handbooks in Operations Research and Management Science*, pages 559–600. Elsevier, 2005.
- Léo Boisvert, Hélène Verhaeghe, and Quentin Cappart. Towards a generic representation of combinatorial problems for learning-based approaches. In *International Conference on the Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, pages 99–108. Springer, 2024.
- Karsten Borgwardt, Elisabetta Ghisu, Felipe Llinares-López, Leslie O’Bray, and Bastian Rieck. Graph kernels: State-of-the-art and future challenges. *Foundations and Trends in Machine Learning*, 13(5-6):531–712, 12 2020.
- Stephen Boyd and Jacob Mattingley. Branch and bound methods. Notes for EE364b, Stanford University, Winter 2006–07, 2007.
- D. Bridge, E. O’Mahony, and B. O’Sullivan. Case-based reasoning for autonomous constraint solving. In *Autonomous search*, pages 73–95. Springer, 2012.
- Quentin Cappart, Didier Chételat, Elias B Khalil, Andrea Lodi, Christopher Morris, and Petar Veličković. Combinatorial optimization and reasoning with graph neural networks. *Journal of Machine Learning Research*, 24(130):1–61, 2023.
- Dillon Z Chen, Felipe Trevizan, and Sylvie Thiébaux. Return to tradition: Learning reliable heuristics with classical machine learning. In *Proceedings of the International Confer-*

- ence on Automated Planning and Scheduling, volume 34, pages 68–76, 2024.
- Geoffrey Chu, Peter J. Stuckey, Andreas Schutt, Thorsten Ehlers, Graeme Gange, and Kathryn Francis. Chuffed: The chuffed cp solver.
- Teodor Gabriel Crainic, Bertrand Le Cun, and Catherine Roucairol. Parallel branch-and-bound algorithms. In *Parallel Combinatorial Optimization*, pages 1–28. John Wiley & Sons, 2006.
- Ian P Gent, Ian Miguel, Peter Nightingale, Ciaran McCreesh, Patrick Prosser, Neil CA Moore, and Chris Unsworth. A review of literature on parallel constraint solving. *Theory and Practice of Logic Programming*, 18(5-6):725–758, 2018.
- Google OR-Tools. CP-SAT solver. Official OR-Tools documentation. Accessed 5 October 2026.
- Leo Katz. A new status index derived from sociometric analysis. *Psychometrika*, 18(1):39–43, 1953.
- Pascal Kerschke, Holger H Hoos, Frank Neumann, and Heike Trautmann. Automated algorithm selection: Survey and perspectives. *Evolutionary computation*, 27(1):3–45, 2019.
- Elias B Khalil, Christopher Morris, and Andrea Lodi. Mipgmn: A data-driven framework for guiding combinatorial solvers. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 36, pages 10219–10227, 2022.
- Sandra Kiefer. *Power and limits of the Weisfeiler-Leman algorithm*. PhD thesis, RWTH Aachen University, 2020.
- Lars Kotthoff. Algorithm selection for combinatorial search problems: A survey. In *Data mining and constraint programming: Foundations of a cross-disciplinary approach*, pages 149–190. Springer, 2016.
- Joao Marques-Silva, Inês Lynce, and Sharad Malik. Conflict-driven clause learning sat solvers. In *Handbook of Satisfiability*, pages 131–153. IOS Press, 2009.
- Ruben Martins, Vasco Manquinho, and Inês Lynce. An overview of parallel sat solving. *Constraints*, 17(3):304–347, 2012.
- Christopher Morris, Yaron Lipman, Haggai Maron, Bastian Rieck, Nils M Kriege, Martin Grohe, Matthias Fey, and Karsten Borgwardt. Weisfeiler and leman go machine learning: The story so far. *Journal of Machine Learning Research*, 24(333):1–59, 2023.
- Mark E. J. Newman. *Networks: An Introduction*. Oxford University Press, 2010.
- Stefan Nickel, Claudius Steinhardt, Hans Schlenker, and Wolfgang Burkart. *Decision optimization with IBM ILOG CPLEX optimization studio*. Springer, Berlin/Heidelberg; Germany, 2022.
- Christos H. Papadimitriou. *Computational complexity*. Addison-Wesley, 1994.
- Francesca Rossi, Peter van Beek, and Toby Walsh, editors. *Handbook of constraint programming*. Elsevier, 2006.
- Nino Shervashidze, Pascal Schweitzer, Erik Jan Van Leeuwen, Kurt Mehlhorn, and Karsten M Borgwardt. Weisfeiler-lehman graph kernels. *Journal of Machine Learning Research*, 12(77):2539–2561, 2011.
- Martin Simonovsky and Nikos Komodakis. Dynamic edge-conditioned filters in convolutional neural networks on graphs. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 3693–3702, 2017.
- Matteo Togninalli, Elisabetta Ghisu, Felipe Llinares-López, Bastian Rieck, and Karsten Borgwardt. Wasserstein weisfeiler-lehman graph kernels. In *Advances in neural information processing systems*, volume 32, 2019.
- S. V. N. Vishwanathan, Nicol N. Schraudolph, Risi Kondor, and Karsten M. Borgwardt. Graph kernels. *The Journal of Machine Learning Research*, 11:1201–1242, 2010.
- Duncan J Watts and Steven H Strogatz. Collective dynamics of ‘small-world’ networks. *nature*, 393(6684):440–442, 1998.
- David H Wolpert and William G Macready. No free lunch theorems for optimization. *IEEE transactions on evolutionary computation*, 1(1):67–82, 1997.
- L. Xu, F. Hutter, J. Shen, H. H Hoos, and K. Leyton-Brown. SATzilla2012: improved algorithm selection based on cost-sensitive classification models. *Proceedings of SAT Challenge*, pages 57–58, 2012.
- Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. How powerful are graph neural networks? *arXiv preprint arXiv:1810.00826*, 2018.
- James Young. Literature review: Graph kernels in chemoinformatics. *arXiv preprint arXiv:2208.04929*, 2022.

Algorithm 1 WLC-n Algorithm for Feature Extraction.

Require: Directed graph $G = (V, E)$, iterations $T \geq 1$, cut nodes $V_{cut} \subseteq V$.

Ensure: Feature histograms of colours and pairs.

```
1: Initialisation:  $c^{(0)}(v) \leftarrow \text{HASH}(\text{type}(v) \parallel \text{subtype}(v))$  for all  $v \in V$ 
2:  $t \leftarrow 0$ 
3: repeat
4:    $t \leftarrow t + 1$ 
5:   for each node  $v \in V$  do
6:     if  $v \notin V_{cut}$  then
7:        $M(v) \leftarrow \{c^{(t-1)}(w) \mid (w, v) \in E\}$ 
8:        $c^{(t)}(v) \leftarrow \text{HASH}(c^{(t-1)}(v), \text{sort}(M(v)))$ 
9:     else
10:       $c^{(t)}(v) \leftarrow c^{(t-1)}(v)$ 
11:    end if
12:  end for
13: until  $t = T$ 
14:  $Pairs \leftarrow \{\}$  ▷ Multiset for pair occurrences
15: for each node  $v \in V_{cut}$  do
16:    $S(v) \leftarrow \{c^{(0)}(w) \mid (w, v) \in E\}$  ▷ Set of initial incoming neighbour types
17:    $c^{(final)}(v) \leftarrow \text{HASH}(c^{(T)}(v), \text{sort}(S(v)))$ 
18:   for each node  $w$  where  $(w, v) \in E$  do
19:      $Pairs \leftarrow Pairs \cup \{(c^{(0)}(w), c^{(0)}(v))\}$ 
20:   end for
21: end for
22:  $c^{(final)}(v) \leftarrow c^{(T)}(v)$  for all  $v \in V \setminus V_{cut}$ 
23: return  $\text{Histogram}(c^{(final)}) \parallel \text{Histogram}(Pairs)$ 
```

A Node Types for FlatZinc

Table 3 shows all the node types used to decompose FlatZinc models to graphs. While these nodes have been especially designed around the FlatZinc interface, we believe that they could easily generalise to different languages, with the only exception of global nodes, which are tied to those supported by the solver used for the MiniZinc to FlatZinc conversion (Gecode in our case).

B The Cut Algorithm

Algorithm 1 outlines the feature extraction procedure for WLC-n. The edge-enhanced variant WLC-ne operates identically, except that edge types are included alongside neighbour colours during aggregation (Line 7).

The execution of Algorithm 1 operates as follows:

- **Line 1 (Initialisation):** Each node $v \in V$ receives an initial colour $c^{(0)}(v)$ based on its type and subtype (e.g., variable, parameter, or operator/constraint type).
- **Lines 3–14 (Iterative Refinement):** Standard Weisfeiler-Leman refinement runs for T steps over non-cut nodes ($v \notin V_{cut}$), updating their colour by hashing the sorted multiset $M(v)$ of incoming neighbour colours (Lines 7–8). Cut nodes skip this aggregation (Line 6), preventing degree-dependent updates, but continue to propagate their colours downstream.
- **Lines 15–19 (Cut Node Aggregation & Pair Counting):** After refinement, cut nodes aggregate the *set*

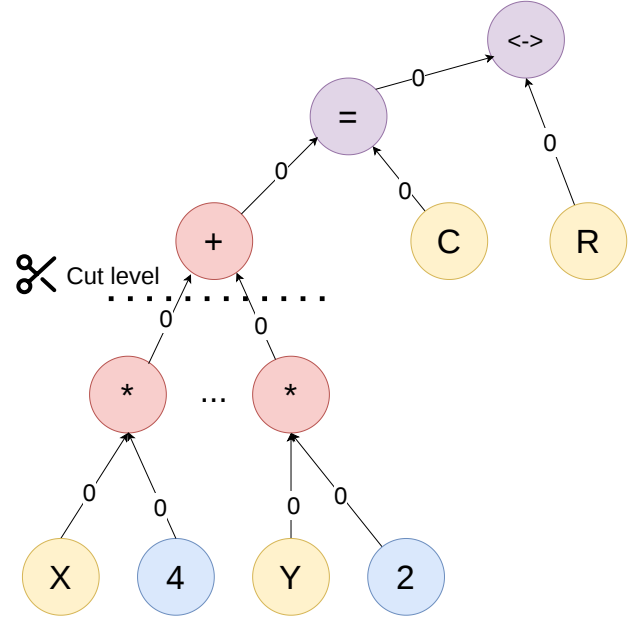


Figure 9: Representation of the deepest constraint currently present in the graph conversion. The constraint corresponds to $\sum_i var_i \times par_i = C \leftrightarrow R$. Variables are in yellow, parameters in blue, operators in red and constraints in purple.

(rather than multiset) $S(v)$ of their incoming neighbour types (Line 16), updating their final colour in Line 17. This ensures cut nodes of the same type with the same set of incoming node types receive the same colour regardless of arity. Lines 18–19 recover the discarded degree information by collecting directed (*neighbour, cut*) pairs into *Pairs*.

- **Lines 22–23 (Feature Extraction):** Non-cut nodes retain their refined colours $c^{(T)}(v)$, and the final feature vector is formed by concatenating the histograms of node colours and neighbour pairs.

C On The Shallowness of the Graphs

Figure 9 shows the constraint that produces the deepest path currently implemented. A family of similar constraints is present, and they usually differ in the type of constraint that replaces the equality (e.g. inequality, less than or equal to, etc.) This local decomposition has a longest directed path of 4 edges, with effective propagation depth reduced to 2 when the cut is applied. Most supported constraints are, instead, quite simple, with most of them having directed paths of at most one or two edges. For the full list of supported constraints, we refer to the FlatZinc specification.⁷

D All Borda Count Results

Figures 10 and 11 show the results on the Borda count tasks for (respectively) RF and MLP classifiers trained with all Fea-

⁷<https://docs.minizinc.dev/en/stable/lib-flatzinc.html>

Name	Type	Subtype	Is Cut	Is Global	Description
Variable	variable	-	No	No	Problem variable
Parameter	parameter	-	No	No	Literal value or parameter
Multiply	operator	mul_node	No	No	Multiplication operator
Linear Sum	operator	lin_sum_node	Yes	No	Weighted sum of variables
Sum	operator	sum_node	No	No	Sum operator
Equality	constraint	equality_node	No	No	Equality constraint
Inequality	constraint	inequality_node	No	No	Inequality constraint
Less than	constraint	le_node	No	No	Strictly less than constraint
Less or Equal	constraint	leq_node	No	No	Less than or equal constraint
ImPLY	constraint	imply_node	No	No	Logical implication
If and only if	constraint	iff_node	No	No	Logical equivalence
Not	constraint	not_node	No	No	Logical negation
Or	constraint	or_node	No	No	Logical disjunction
And	constraint	and_node	No	No	Logical conjunction
Xor	constraint	xor_node	No	No	Logical exclusive or
Index	operator	index_node	No	No	Array index operator
Abs	operator	abs_node	No	No	Absolute value operator
Division	operator	division_node	No	No	Integer division operator
Max	operator	max_node	No	No	Maximum operator
Min	operator	min_node	No	No	Minimum operator
Modulo	operator	modulo_node	No	No	Modulo operator
Pow	operator	pow_node	No	No	Power operator
In	constraint	in_node	No	No	Set membership constraint
Card	operator	card_node	No	No	Set cardinality operator
Difference	operator	diff_node	No	No	Set difference operator
Intersect	operator	intersect_node	No	No	Set intersection operator
Subset	constraint	subset_node	No	No	Set subset constraint
Symmetric difference	operator	symdiff_node	No	No	Set symmetric difference operator
Union	operator	union_node	No	No	Set union operator
Maximise	solve	maximise_node	No	No	Maximisation objective
Minimise	solve	minimise_node	No	No	Minimisation objective
Array	operator	array_node	No	No	Array container
Cumulative	constraint	cumulative_node	Yes	Yes	Global cumulative constraint
Integer Element	operator	int_element_node	Yes	Yes	Global element constraint
Linear equality iff	constraint	int_lin_eq_imp_node	Yes	No	Reified linear equality
Array max	constraint	array_int_maximum_node	Yes	Yes	Global array maximum constraint
Schedule unary	constraint	schedule_unary_node	Yes	Yes	Global unary scheduling constraint
Linear less than iff	constraint	int_le_imp_node	Yes	No	Reified linear less than
Maximum argument offset	constraint	maximum_arg_int_offset_node	Yes	Yes	Global maximum argument constraint
Circuit	constraint	circuit_node	Yes	Yes	Global circuit constraint
Count equality	constraint	count_eq_node	Yes	Yes	Global count equality constraint
Count equality iff	constraint	count_eq_reif_node	Yes	Yes	Reified count equality constraint
Set in iff	constraint	set_in_imp_node	Yes	Yes	Reified set membership constraint
Global cardinality	constraint	global_cardinality_node	Yes	Yes	Global cardinality constraint
Global cardinality low up	constraint	global_cardinality_low_up_node	Yes	Yes	Global cardinality constraint with bounds
Global cardinality low up closed	constraint	global_cardinality_low_up_closed_node	Yes	Yes	Global cardinality constraint with closed bounds
Xor iff	constraint	bool_xor_imp_node	Yes	No	Reified exclusive or constraint
No overlap	constraint	nooverlap_node	Yes	Yes	Global no overlap constraint
Regular	constraint	regular_node	Yes	Yes	Global regular constraint
All different	constraint	all_different_node	Yes	Yes	Global all different constraint
Equality iff	constraint	eq_imp_node	Yes	Yes	Reified equality constraint
All equal	constraint	all_equal_node	Yes	Yes	Global all equal constraint
Boolean Element	operator	bool_element_node	Yes	No	Global element constraint (boolean)
Array minimum	constraint	array_int_minimum_node	Yes	No	Global array minimum constraint
Bool clause iff	constraint	bool_clause_reif_node	Yes	No	Reified boolean clause constraint
Element 2d	operator	int_element2d_node	Yes	Yes	Global 2D element constraint
Bin packing load	constraint	bin_packing_load_node	Yes	Yes	Global bin packing load constraint
Table	constraint	table_int_node	Yes	Yes	Global table constraint
Precede	constraint	precede_node	Yes	Yes	Global precede constraint
Array less or equal	constraint	array_int_lq_node	Yes	No	Global array less or equal constraint
Linear less than iff	constraint	int_lin_le_imp_node	Yes	No	Reified linear less than
Linear not equal iff	constraint	int_lin_ne_imp_node	Yes	No	Reified linear not equal
Increasing integer	constraint	increasing_int_node (int)	Yes	Yes	Global increasing integer constraint
Inverse	constraint	inverse_offsets_node	Yes	Yes	Global inverse constraint
N value	constraint	nvalue_node	Yes	Yes	Global n-value constraint
Not equal iff	constraint	int_ne_imp_node	Yes	No	Reified not equal constraint
Increasing bool	constraint	increasing_bool_node	Yes	Yes	Global increasing boolean constraint
Member integer	constraint	member_int_node	Yes	Yes	Global member integer constraint
Table iff	constraint	table_int_imp_node	Yes	Yes	Reified global table constraint
At least	constraint	at_least_node	Yes	Yes	Global at least constraint
At most	constraint	at_most_node	Yes	Yes	Global at most constraint
Integer pow	constraint	int_pow_node	Yes	Yes	Global integer power constraint
Global cardinality closed	constraint	global_cardinality_closed_node	Yes	Yes	Global cardinality constraint closed

Table 3: Node types used in our graphs converted from FlatZinc.

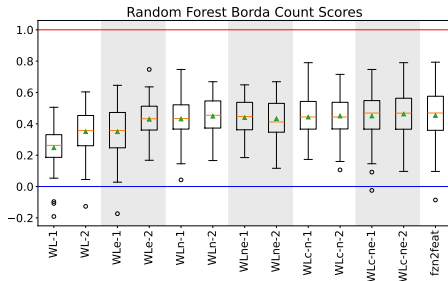


Figure 10: Results of all feature sets trained with the RF classifier on the Borda count maximisation task.

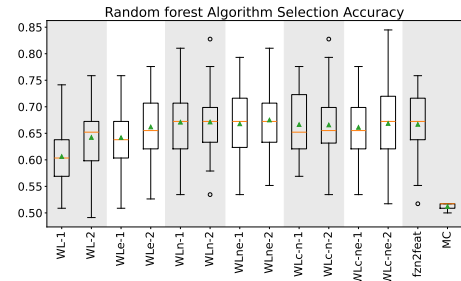


Figure 12: Results of all feature sets trained with the RF classifier on the Algorithm Selection accuracy maximisation task.

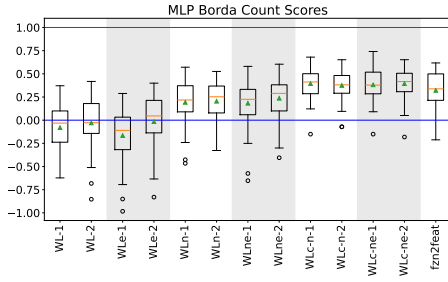


Figure 11: Results of all feature sets trained with the MLP classifier on the Borda count maximisation task.

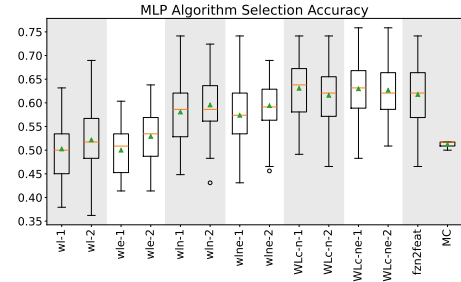


Figure 13: Results of all feature sets trained with the MLP classifier on the Algorithm Selection accuracy maximisation task.

ture sets. RF results show similar results for all WL based features that include node information, as well as `fzn2feat` features. On the other hand, MLP results show a trend similar to SVM, where cut features significantly outperform the other WL-based features, as well as showing a higher gap with respect to `fzn2feat`.

E All Algorithm Selection Accuracy Results

Figures 12 and 13 show the results on the Algorithm selection accuracy tasks for (respectively) RF and MLP classifiers trained with all Feature sets. Similarly to the Borda count counterpart, RF results show similar results for all WL based features that include node information as well as `fzn2feat` features, while MLP results show a trend similar to SVM where cut features significantly outperform the other WL-based features as well as showing a higher gap with respect to `fzn2feat` (although less pronounced than in the Borda count task).

F Ablation Study on the Importance of Concatenating All Aggregation Levels

To assess whether only using the last aggregation level of the WL resulted in a loss of performance, we conducted an ablation study on the best-performing features: `WLC-n-1/2`, `WLC-ne-1/2`, `WLn-1/2`, and `WLne-1/2` across all different tasks and ML models. Results can be found in Tables 4 (Borda score), 5 (algorithm selection accuracy), and 6 (parallel prediction). The tables show the mean and median results of the ML models trained on only the last level or on all levels of aggregation, as well as the p-value computed

with the Wilcoxon test. In all cases but one, the results are not statistically significant. The only case in which they are (`WLC-n-1` in SVM algorithm selection accuracy prediction), it is in favour of the features with only the last aggregation level.

(a) Results of SVM models.

	Last mean	All-levels mean	Last median	All-levels median	p value
WL ₀ -1	0.210	0.196	0.245	0.237	0.632
WL ₀ -2	0.283	0.243	0.276	0.266	0.357
WL _{0a} -1	0.226	0.212	0.258	0.245	0.561
WL _{0a} -2	0.276	0.236	0.275	0.266	0.201
WL _{0c} -1	0.485	0.441	0.506	0.473	0.399
WL _{0c} -2	0.472	0.432	0.504	0.456	0.198
WL _{0c-na} -1	0.470	0.440	0.513	0.487	0.681
WL _{0c-na} -2	0.479	0.432	0.475	0.460	0.318

(b) Results of RF models.

	Last mean	All-levels mean	Last median	All-levels median	p value
WL ₀ -1	0.431	0.464	0.434	0.486	0.318
WL ₀ -2	0.449	0.464	0.454	0.465	0.745
WL _{0a} -1	0.440	0.452	0.447	0.455	0.509
WL _{0a} -2	0.433	0.475	0.412	0.488	0.114
WL _{0c} -1	0.443	0.460	0.444	0.465	0.608
WL _{0c} -2	0.450	0.449	0.444	0.459	0.863
WL _{0c-na} -1	0.450	0.456	0.469	0.484	0.774
WL _{0c-na} -2	0.462	0.450	0.468	0.445	0.472

(c) Results of MLP models.

	Last mean	All-levels mean	Last median	All-levels median	p value
WL ₀ -1	0.192	0.212	0.192	0.218	0.229
WL ₀ -2	0.204	0.201	0.252	0.229	0.826
WL _{0a} -1	0.183	0.182	0.225	0.217	0.811
WL _{0a} -2	0.236	0.187	0.289	0.237	0.237
WL _{0c} -1	0.364	0.397	0.412	0.420	0.910
WL _{0c} -2	0.375	0.374	0.381	0.386	0.695
WL _{0c-na} -1	0.381	0.402	0.378	0.426	0.534
WL _{0c-na} -2	0.394	0.391	0.417	0.418	0.826

Table 4: Results of the ML models trained for Borda count maximisation on WL features with (all-levels) and without (last) all aggregation levels. Results report mean and median scores, as well as the p-value calculated with the Wilcoxon test.

(a) Results of SVM models.

	Last mean	All-levels mean	Last median	All-levels median	p value
WL ₀ -1	0.589	0.596	0.591	0.600	0.364
WL ₀ -2	0.607	0.591	0.603	0.603	0.217
WL _{0a} -1	0.589	0.597	0.600	0.596	0.336
WL _{0a} -2	0.605	0.604	0.603	0.603	0.974
WL _{0c} -1	0.695	0.663	0.681	0.655	0.030
WL _{0c} -2	0.687	0.664	0.690	0.672	0.089
WL _{0c-na} -1	0.693	0.674	0.690	0.664	0.251
WL _{0c-na} -2	0.690	0.659	0.690	0.661	0.060

(b) Results of RF models.

	Last mean	All-levels mean	Last median	All-levels median	p value
WL ₀ -1	0.671	0.678	0.672	0.672	0.584
WL ₀ -2	0.671	0.677	0.672	0.672	0.611
WL _{0a} -1	0.668	0.674	0.672	0.672	0.804
WL _{0a} -2	0.675	0.672	0.672	0.672	0.866
WL _{0c} -1	0.666	0.669	0.652	0.655	0.615
WL _{0c} -2	0.666	0.661	0.655	0.655	0.619
WL _{0c-na} -1	0.661	0.665	0.661	0.655	0.930
WL _{0c-na} -2	0.668	0.659	0.672	0.655	0.322

(c) Results of MLP models.

	Last mean	All-levels mean	Last median	All-levels median	p value
WL ₀ -1	0.581	0.577	0.586	0.586	0.841
WL ₀ -2	0.596	0.594	0.586	0.603	0.714
WL _{0a} -1	0.574	0.581	0.574	0.583	0.365
WL _{0a} -2	0.594	0.586	0.591	0.603	0.532
WL _{0c} -1	0.631	0.635	0.638	0.621	0.828
WL _{0c} -2	0.616	0.616	0.621	0.621	0.841
WL _{0c-na} -1	0.630	0.630	0.632	0.626	0.935
WL _{0c-na} -2	0.627	0.616	0.621	0.621	0.571

Table 5: Results of the ML models trained for algorithm selection accuracy maximisation on WL features with (all-levels) and without (last) all aggregation levels. Results report mean and median scores, as well as the p-value calculated with the Wilcoxon test.

(a) Results of SVM models.

	Last mean	All-levels mean	Last median	All-levels median	p value
WL ₀ -1	0.627	0.620	0.633	0.617	0.935
WL ₀ -2	0.611	0.626	0.605	0.617	0.327
WL _{0a} -1	0.618	0.627	0.633	0.622	0.295
WL _{0a} -2	0.625	0.628	0.622	0.625	0.851
WL _{0c} -1	0.625	0.619	0.617	0.633	0.615
WL _{0c} -2	0.604	0.615	0.600	0.617	0.438
WL _{0c-na} -1	0.633	0.610	0.627	0.617	0.995
WL _{0c-na} -2	0.604	0.601	0.600	0.600	0.845

(b) Results of RF models.

	Last mean	All-levels mean	Last median	All-levels median	p value
WL ₀ -1	0.642	0.651	0.633	0.650	0.409
WL ₀ -2	0.634	0.638	0.650	0.658	0.658
WL _{0a} -1	0.643	0.649	0.633	0.661	0.619
WL _{0a} -2	0.654	0.645	0.650	0.633	0.435
WL _{0c} -1	0.635	0.645	0.650	0.650	0.369
WL _{0c} -2	0.645	0.650	0.644	0.650	0.476
WL _{0c-na} -1	0.643	0.634	0.647	0.633	0.464
WL _{0c-na} -2	0.642	0.648	0.650	0.650	0.631

(c) Results of MLP models.

	Last mean	All-levels mean	Last median	All-levels median	p value
WL ₀ -1	0.621	0.627	0.633	0.633	0.975
WL ₀ -2	0.622	0.606	0.630	0.608	0.097
WL _{0a} -1	0.618	0.620	0.622	0.633	0.854
WL _{0a} -2	0.618	0.616	0.630	0.613	0.952
WL _{0c} -1	0.644	0.645	0.633	0.644	0.886
WL _{0c} -2	0.653	0.660	0.650	0.667	0.597
WL _{0c-na} -1	0.649	0.646	0.650	0.650	0.695
WL _{0c-na} -2	0.645	0.662	0.647	0.667	0.074

Table 6: Results of the ML models trained for parallel prediction on WL features with (all-levels) and without (last) all aggregation levels. Results report mean and median scores, as well as the p-value calculated with the Wilcoxon test.