

École polytechnique de Louvain

Prediction-based Propagation in Constraint Programming

A Focus on the AllDifferent Constraint

Author: **Corentin DERUYCK**
Supervisor: **Hélène VERHAEGHE**
Readers: **Charles PACKO, Amaury GUICHARD, Quentin CAPPART**
Academic year 2025–2026
Master [120] in Computer Science and Engineering

Abstract

Constraint programming is a very interesting approach when one is looking for a set of correct solutions to a problem that can be modeled as variables with specific domains linked by constraints. However, for many problems, the time required to find these solutions, as well as the size of the search tree, can present a significant challenge.

This thesis aims to build a proof of concept by constructing an *AllDifferent* propagator based on machine learning in order to simulate the behavior of the *Domain Consistency* propagator. The goal is, through a variable threshold that modifies the model's behavior, to more or less reduce the search tree and thereby find solutions more quickly that are uniformly distributed within this tree.

The approach also applies to constraint optimization problems (COPs), in order to assess whether the constructed model is capable of finding a solution, subject to an optimization function, that is sufficiently close to the best solution obtained using a classical solver.

Acknowledgments

I would like to begin by thanking my supervisor, H       Verhaeghe, for her remarkable availability, her numerous pieces of advice, and her constant encouragement. Her support helped me stay on track until the end of my final-year project. I am grateful for the richness of our discussions every two weeks, from the initial inconclusive results to the enthusiasm sparked by the more promising ones.

I would also like to thank Charles Packo for his help and for his insightful comments regarding my poster, my code, and my written work. His questions about my project stimulated my thinking and helped enrich the entire undertaking. I wish him much success as he continues his PhD, hoping that this first building block will provide the inspiration he needs to pursue this line of research.

Finally, I would like to thank my friends, my family, and my girlfriend, who took the time to understand my work, listen to my concerns, and offer me valuable advice. It is thanks to them that, despite the challenges, these years at university will remain a precious memory for me, forever engraved in my mind.

The ChatGPT generation tool was used to proofread the English translation of this master's thesis.

Contents

1	Introduction	7
2	Theoretical Framework	8
2.1	Constraint Programming Paradigm	8
2.1.1	Definition	8
2.1.2	Solving a CSP	9
2.2	The AllDifferent Constraint	11
2.2.1	Arc Consistency	11
2.2.2	Domain Consistency	12
3	Machine Learning	18
3.1	Multilayer Perceptron	18
3.2	Graph Neural Network	20
4	State of the Art	21
4.1	Large Neighborhood Search	22
4.2	Contrastive Predict-and-Search	22
5	Propagator Construction	24
5.1	Solver	24
5.2	Data	24
5.2.1	Instance Selection	24
5.2.2	Solution Density	25
5.2.3	Data Collection Before and After Propagation	26
5.3	Machine Learning	30
5.3.1	Data Representation	30
5.3.2	Model Architecture	31
5.4	Integration into the Propagator	32
5.4.1	Java–Python Communication	32
5.4.2	Checker	33
6	Model Creation	36
6.1	Baseline Model	36
6.2	Small Model	37
6.3	Random Model Sampled from the Baseline Distribution	38

6.4	Fully Random Model	40
7	Experiments	41
7.1	Evaluation Outside the Propagator	41
7.1.1	Definition of Metrics	41
7.1.2	Practical Evaluation	43
7.2	Number of Solutions	51
7.2.1	Baseline Model	52
7.2.2	Small Model	55
7.2.3	Random Model Sampled from the Baseline Distribution	56
7.2.4	Fully Random Model	57
8	Runtime Analysis	59
8.1	Hardware Configuration	59
8.2	Runtime Measurement	59
9	Constraint Optimization Problems	62
9.1	Theory	62
9.2	Problems Under Study	63
9.3	Experiments	66
9.3.1	Baseline Model	66
9.3.2	Small Model	68
9.3.3	Random Model Sampled from the Baseline Distribution	70
9.3.4	Fully Random Model	72
10	Limitations and Perspectives	76
10.1	Limitations	76
10.2	Future Research Perspectives	76
11	Conclusion	78
	<i>References</i>	79
	Appendices	82

Chapter 1

Introduction

Constraint programming is an excellent paradigm to use when it comes to solving problems that can be modeled in the form of variables with domains and linked by constraints. The basic method proposes the construction of a search tree to find a set of solutions satisfying the constraints. Moreover, in order to reduce the search tree, custom propagators are linked to each constraint. Their goal is to remove as quickly as possible the domain values that cannot lead to a solution.

This thesis focuses on the propagator of the *AllDifferent* constraint. The most efficient propagation algorithm for this constraint is based on Régin's algorithm. Although this algorithm is very effective, it is rather slow. This thesis therefore aims to build a machine learning model that attempts to reproduce the behavior of the propagation algorithm. One point of interest is to determine whether the propagation time is faster with the machine learning algorithm or with the classical algorithm. Another point of interest is to use a variable threshold so that the algorithm can make more or less risky decisions when removing values from the domains. In this sense, a very conservative threshold will keep a large number of nodes and therefore a large number of solutions, whereas an aggressive threshold will reduce the search tree and possibly the number of solutions.

The method also targets constraint optimization problems. In these problems, not all solutions are equally good. Indeed, an objective function assigns a score to each solution, and the goal is to find the solution that maximizes or minimizes this score. The idea is therefore to use our machine learning model with a variable threshold to measure whether the best solution obtained is as good as the best solution found with the classical propagator.

This thesis could be very useful for certain constraint programming problems that generate a very large search tree and for which a subset of solutions, or a solution close to the best one, is sufficient. There are thus two ways to consider the approach as interesting. Either by building a model whose behavior is close to the classical algorithm but with a reduced execution time, or by obtaining a model that variably reduces the size of the search tree while still producing high-quality solutions in the case of optimization problems.

Chapter 2

Theoretical Framework

2.1 Constraint Programming Paradigm

2.1.1 Definition

Constraint programming is a programming paradigm that allows a user to specify a constraint satisfaction problem (CSP) by defining the problem variables, the domain of these variables, and the constraints linking the variables together. The solver will then find different solutions that satisfy the problem. A valid solution is an assignment of a variable to a value within its domain that respects the imposed constraints. A problem is said to be infeasible if it has no solution that satisfies the constraints [1].

More formally, a constraint satisfaction problem (CSP) can be defined as a triplet $P = (X, D, C)$ with

- $X = \{X_1, X_2, \dots, X_n\}$ a set of variables.
- $D = \{D_1, D_2, \dots, D_n\}$ the different domains, with D_i being the domain of variable X_i .
- $C = \{C_1, C_2, \dots, C_m\}$ a set of constraints.

Example 2.1. *To illustrate the definitions, let us present the Hello World of CSP problems: the Four Queens problem. The goal is to place four queens on a 4×4 chessboard while ensuring that the four queens do not attack each other. To achieve this, we know that we must place one queen in each column of the chessboard, otherwise, two queens would be placed in the same column and could attack each other. The problem can be formulated as follows.*

- $X = \{X_1, X_2, X_3, X_4\}$, where X_i is the row in which the queen will be placed in column i .
- $D_i = \{1, 2, 3, 4\}$, a queen in column i can be placed in any row from 1 to 4.
- *Constraints:*
 - Two queens cannot be on the same row: $X_i \neq X_j$ for $i \neq j$.
 - Two queens cannot be on the same diagonal: $|X_i - X_j| \neq |i - j|$ for $i \neq j$.

Note that the constraint preventing two queens from being placed in the same column does not need to be defined due to the way the problem is modeled [2].

The problem, as defined above, has two solutions illustrated in Figure 2.1.

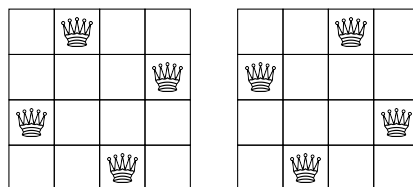


Figure 2.1: The two solutions to the four-queens problem

2.1.2 Solving a CSP

Post and Propagate

When a user wants to link several variables through a constraint, they must create an object representing the constraint by specifying the different variables involved. A constraint object contains the *post* method and the *propagate* method. When the constraint is given to the solver, it calls the *post* method. This method has several roles, such as initializing the class variables used by the constraint, calling the *propagate* method for the first time, and specifying for each variable when the *propagate* method should be called. For example, the *propagate* method can be called whenever there is a change in the variable's domain, when the bounds of the variable's domain change, or when the variable is assigned a unique value.

The role of the *propagate* method is to reduce the size of the variable domains based on an algorithm. The algorithm can be more or less complex in order to identify and remove values that cannot be part of a solution according to the constraint. As section 2.2 on the *AllDifferent* constraint will illustrate, a choice must be made between a complex (and sometimes slower) algorithm that eliminates all values that can in no case be part of a solution, and a faster algorithm that removes fewer values.

Search tree

To find a solution, a search tree is constructed. This tree makes a decision at each step to remove one or more values from the domains. For example, a decision at a node of the tree can be to choose a variable and create two branches. On one side of the tree, the variable is assigned a value from its domain, and on the other side, the value is removed from the domain. The choice of the variable at each node of the tree, as well as the choice of which value to assign, can influence the efficiency of the search for a solution.

By assigning or removing certain values from the domains, some variables may be woken up according to the specification of the constraint's *post* method. In this case, the constraint will be scheduled in a queue of constraints to be processed.

Example 2.2. *If the post method of constraint C_1 specifies that it must be awakened when the domain of variable X_1 changes, and the search tree makes the decision to remove a value from variable X_1 , then constraint C_1 will be added to the queue.*

This queue allows the calculation of the *fixpoint*. To do this, the solver calls the *propagate* method of each constraint in the queue at each step of the search. In this way, some values in the domains may be further removed by the *propagate* method, which will cause the constraint to be rescheduled in the queue. Once the queue is empty, the *fixpoint* has been reached, and a new decision can be made to expand the search tree.

If at any point during the search a variable ends up with an empty domain, or if the *propagate* method finds an inconsistency that can no longer lead to a solution, the current branch of the tree is stopped and backtracked to try another path. This is called backtracking. In this case, all domains are reconstructed while moving back up the tree.

Example 2.3. Consider three variables X_1 , X_2 , and X_3 linked by the *AllDifferent* constraint. This constraint, described in detail in the [section 2.2](#), ensures that each variable involved in the constraint has a different value.

The domains of the variables are as follows:

- $X_1 = \{1, 2\}$
- $X_2 = \{1, 2\}$
- $X_3 = \{1, 2, 3\}$

Let us imagine that the *post* method of the constraint specifies that the constraint should be scheduled whenever the domain of a variable changes. Furthermore, suppose that the *propagate* method simply removes from the domains of unfixed variables the values that are already assigned in fixed variables.

[Figure 2.2](#) shows the search tree leading to two different solutions.

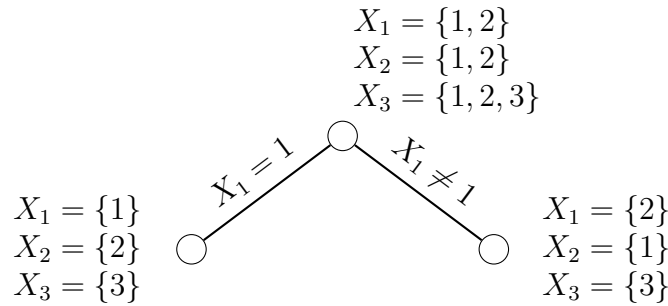


Figure 2.2: Example of a search tree with three variables linked by the *AllDifferent* constraint

First, an initial *fixpoint* can be computed at the root of the tree. This will not remove any values from the domains. Next, a first decision assigns X_1 to 1, creating the first branch in the tree. This decision causes the constraint to be scheduled because the domain of X_1 has changed. The *fixpoint* calculation will select the constraint from the list of scheduled constraints and call the *propagate* method. This method will realize that variable X_1 is fixed to 1, and therefore the value 1 will be removed from the domains of X_2 and X_3 . By doing this, the constraint will be scheduled again because the domains of X_2 and X_3 have changed. The *fixpoint* is thus not complete and will continue to call the *propagate* method of the scheduled constraints. This

time, the propagate method will see that variable X_2 is fixed to 2 and will remove the value 2 from X_3 . At this point, all variables are fixed without violating the constraint, and a first solution is found. The search tree is then backtracked by one level, the domains are restored, and a similar execution occurs by removing 1 from the domain of X_1 .

2.2 The AllDifferent Constraint

The constraint of interest in this thesis is the *AllDifferent* constraint. This constraint allows specifying a set of variables and ensures that the final value of each variable linked to the constraint will be different from the values of the other variables defined within the constraint.

More formally, for a set of variables $X = \{X_1, X_2, \dots, X_n\}$ all linked by the *AllDifferent* constraint, the following relation must be satisfied.

$$\forall i, j \in \{1, \dots, n\}, i \neq j \Rightarrow X_i \neq X_j$$

As stated above, the role of the propagator is to reduce the domains of variables by removing values that cannot be part of a solution with certainty. The *AllDifferent* constraint has different propagation algorithms, each with its own advantages and disadvantages. Two of these propagators are described in [subsection 2.2.1](#) and in [subsection 2.2.2](#).

2.2.1 Arc Consistency

The arc consistency property guarantees that for every value $x \in D_i$ of a variable X_i , there exists at least one value $y \in D_j$ of a variable X_j linked by a constraint to X_i , such that the constraint between variables X_i and X_j is satisfied. In the case of a constraint involving more than two variables, this property must be satisfied for each pair of concerned variables [3,4].

The algorithm used to achieve the arc consistency property is the Forward Checking [5]. It is a simple propagator but has the advantage of being fast. The idea is to remove from the domains of unfixed variables the values that have already been assigned to fixed variables.

Example 2.4. *Let us consider that we reach the following state at some point during the search.*

- $X_1 = \{1, 2, 3\}$
- $X_2 = \{2\}$
- $X_3 = \{1, 2\}$

Since variable X_2 is fixed to the value 2, the value 2 can be removed from the domains of X_1 and X_3 . Next, variable X_3 also becomes fixed to the value 1, allowing the value 1 to be removed from the domain of X_1 . Finally, a solution is obtained: $X_1 = 3$, $X_2 = 2$, and $X_3 = 1$.

Although this propagator is fast, it does not remove all domain values that can never be part of a solution when considering one constraint at a time.

2.2.2 Domain Consistency

The Domain Consistency property (or Generalized Arc Consistency) guarantees that, for every value $x \in D_i$ of a variable X_i , there exists at least one combination of values in the domains of the other variables involved in the constraint such that the constraint is satisfied [4].

Example 2.5. *Let us consider the situation below, reached at some point during the search.*

- $X_1 = \{1, 2\}$
- $X_2 = \{1, 2\}$
- $X_3 = \{1, 3\}$

The Forward Checking propagator would not be able to do anything in this situation. However, it is still possible to remove values from the domain that cannot lead to a solution with certainty. Indeed, in this situation, we see that X_1 and X_2 each have a domain of size two with the same values. We therefore know that either $X_1 = 1$ and $X_2 = 2$, or $X_1 = 2$ and $X_2 = 1$. In this case, it is possible to safely remove the value 1 from the domain of variable X_3 , because it is impossible to reach a solution with X_3 assigned to 1 from this state.

The following lines rely on the article [6] and the course [7] to introduce certain definitions as well as several algorithms.

To satisfy this property for the *AllDifferent* constraint, the propagator represents the variables and their domains as a bipartite graph. On the left side of the graph are the variables, and on the right side are the domain values. An edge connects a variable X_i to a value v_j if X_i has v_j in its domain.

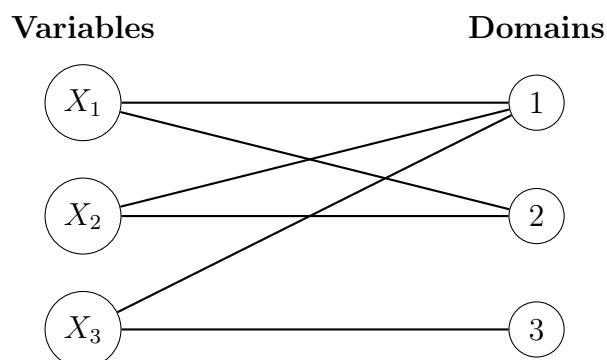


Figure 2.3: Example of representing variables and their domains as a bipartite graph

Based on this, let us introduce a few definitions.

- A **matching** M is a set of edges with no two edges sharing a vertex.
- An edge that belongs to a matching M is called a **matching edge**.
- An edge that does not belong to a matching M is called a **free edge**.

- A vertex incident to a matching M is an **M-matched vertex**.
- A vertex not incident to a matching M is an **M-free vertex**.
- An **alternating path (or cycle)** is a path (or cycle) whose edges alternate between matching and free edges.
- The **length** of an alternating path is the number of edges it contains.
- A **maximal matching** is a matching that cannot be extended by adding another edge.
- The **size** of a matching M is the number of edges in the matching M .

Figure 2.4 illustrates these definitions, with a maximal matching M represented by the edges in blue.

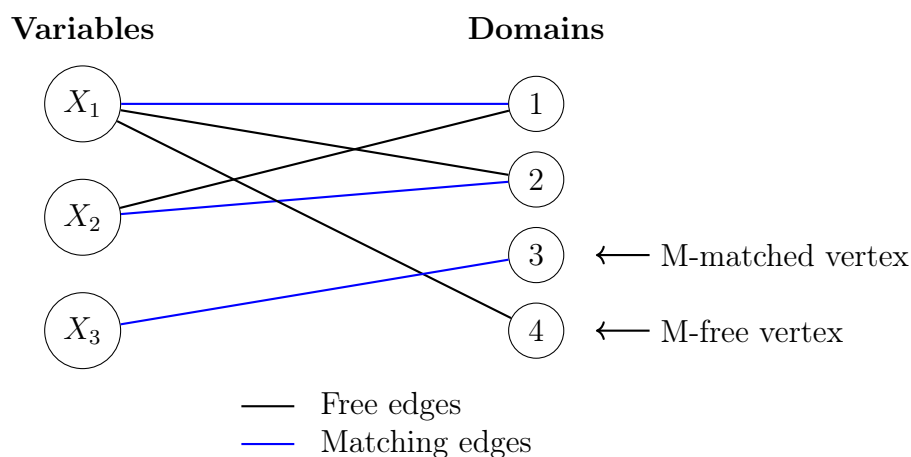


Figure 2.4: Illustration of the definitions in a bipartite graph with a maximal matching M

Figure 2.5 illustrates, in blue, an alternating cycle for the maximal matching shown in Figure 2.4.

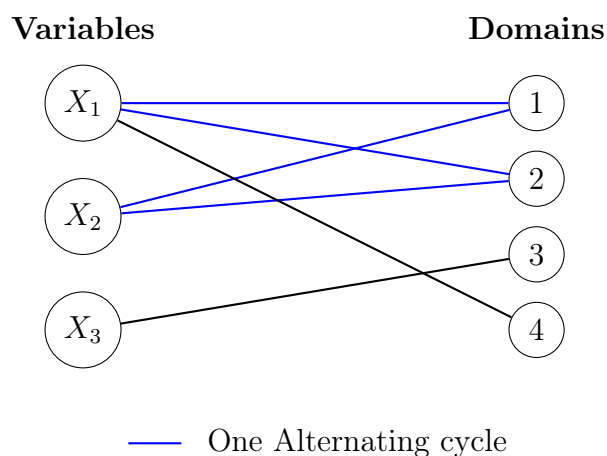


Figure 2.5: Illustration of an alternating cycle based on a maximal matching M

Feasibility Check

The first step is to ensure that each *AllDifferent* constraint, taken independently, does not lead to an inconsistency. This step does not allow one to prove the existence of a solution from this point in the search. However, if an inconsistency is detected, it proves that no solution can be obtained from this stage of the search. Consequently, in the case of infeasibility, the search can be terminated and backtracking can be performed.

To do this, the algorithm must find a maximal matching of size equal to the number of variables. Indeed, if such a maximal matching exists, each variable on the left side of the bipartite graph can be linked to a value in its domain while ensuring a different value for each variable. Figure 2.4, which contains three variables, illustrates such a maximal matching of size three.

The algorithm will start in a greedy manner, beginning with an empty matching and iteratively adding an edge between a variable and a value from its domain. When it is no longer possible to add an edge between a variable and a value from its domain without breaking the matching property, this first phase stops.

Figure 2.6 shows how the greedy algorithm searches for a maximal matching. It starts with variable X_1 , linking it to the value 1 in its domain, then with X_2 linked to the value 3 in its domain. With the selected edges, it is no longer possible to add an edge linking X_3 without breaking the matching property.

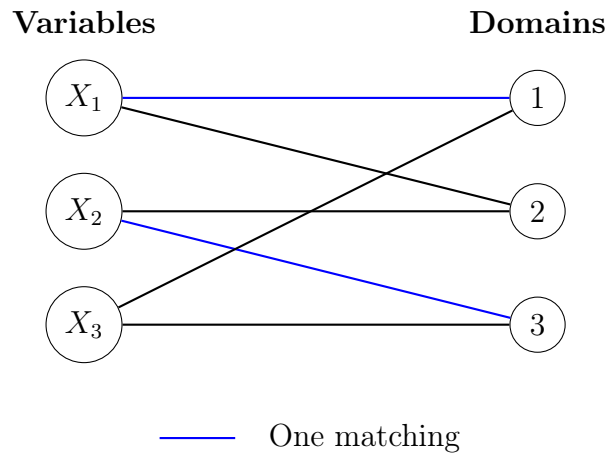


Figure 2.6: Finding a maximal matching using a greedy algorithm

The second phase consists of iteratively modifying the matching to increase its size until a maximal matching of size equal to the number of variables is obtained, or an inconsistency is found. To do this, edges belonging to the matching are oriented from the domains to the variables, while edges not in the matching are oriented from the variables to the domains.

Next, a DFS algorithm starts from an M-free variable vertex and tries to find an alternating path ending at an M-free value vertex. By definition, following the orientation of the edges ensures that the algorithm follows an alternating path.

If, at the end of the DFS search, no alternating path exists, there is an inconsistency for this state. Otherwise, the algorithm will reverse the status of the edges along the alternating

path. If an edge in the alternating path belongs to the matching, it will no longer belong to it, and if it did not belong to the matching, it will become part of it. In this way, the size of the matching is increased by one.

Figure 2.7 illustrates the process of increasing the size of the matching using the bipartite graph from Figure 2.6 as an example. In the left figure, the alternating path is shown with dashed lines and the old matching in blue. In the right figure, the new matching is represented in blue.

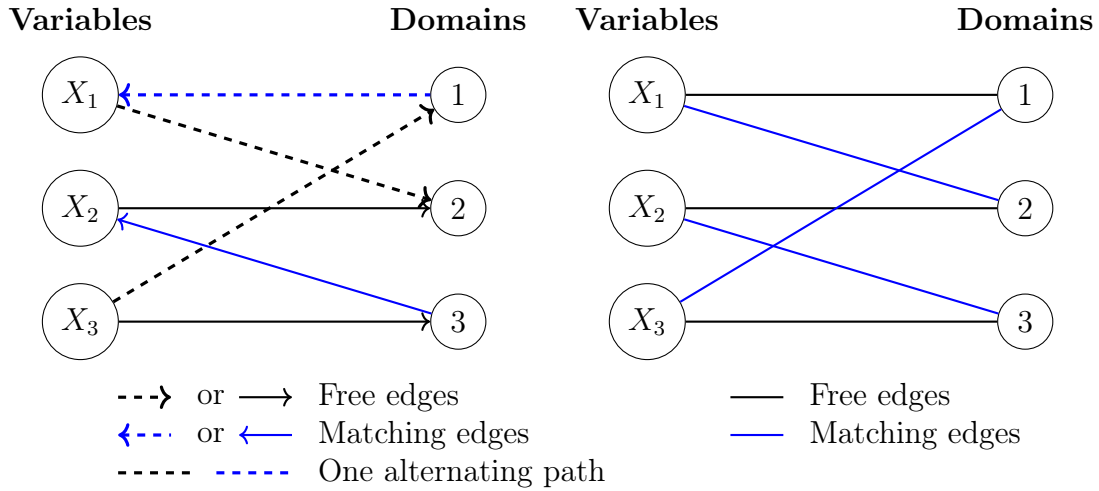


Figure 2.7: Illustration of the iterative algorithm used to increase the matching size through an alternating path

The algorithm continues in this manner until either an inconsistency is found or a maximal matching is obtained. It is worth noting that the described algorithm would also work if the greedy search were skipped and the second phase started directly with an empty matching.

Filtering

In the following, we assume that a valid maximal matching must have a size equal to the number of variables.

To filter the domains, a first idea would be to enumerate all possible maximal matchings in the bipartite graph and then remove the edges that do not belong to any of these maximal matchings. Although this algorithm would work, its computational complexity would be catastrophic.

An alternative approach is based on Berge's theorem [8,9]. This theorem states that an edge belongs to a maximal matching if and only if, for any maximal matching M , the edge either belongs to an alternating path of even length starting at an M -free vertex, or the edge belongs to an alternating cycle of even length.

Figure 2.8 illustrates these two cases. In the first case (on the left), we observe that by flipping the status of the dashed edges corresponding to an alternating path starting from an M -free vertex of even length, the new matching remains a valid maximal matching. The edge from X_2 to 3 therefore indeed belongs to a maximal matching and should not be removed. In

the second case (on the right), we similarly observe that by flipping the status of the dashed edges corresponding to an alternating cycle of even length, we obtain a new valid maximal matching. The edges from X_1 to 1 and from X_2 to 2 thus belong to a maximal matching and should not be removed.

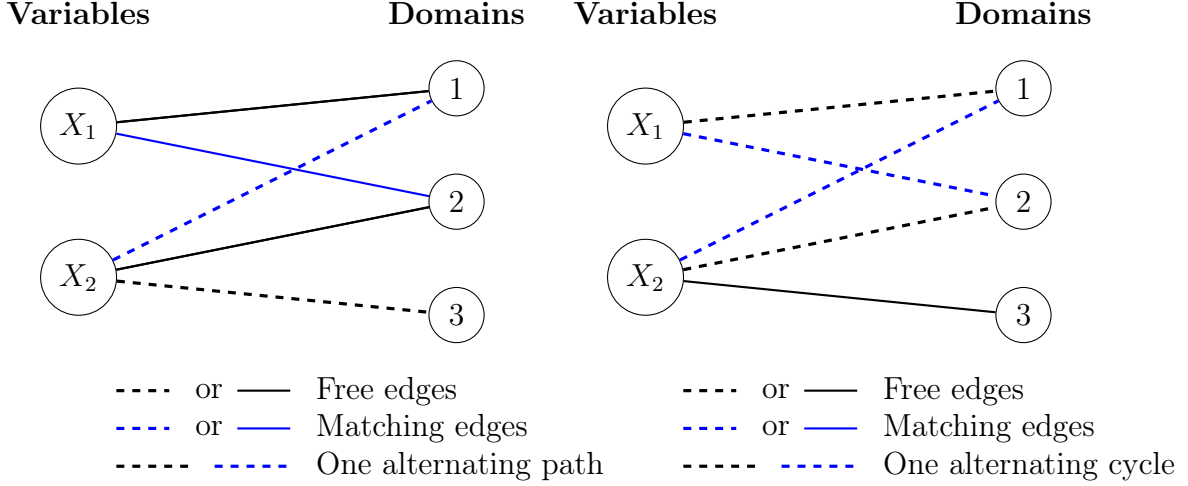


Figure 2.8: Discovery of edges belonging to a maximal matching using the method of even-length alternating paths starting from an M-free vertex and that of even-length alternating cycles

It is possible to combine the two cases into a single one by introducing a dummy node that connects all domain nodes and by orienting the edges. The edges will point from domain nodes to the dummy node for those adjacent to M-free vertices, and from the dummy node to domain nodes for the others. Furthermore, as in the consistency check, the edges belonging to the maximal matching M will be oriented from domain nodes to variable nodes, while the edges not belonging to the maximal matching M will be oriented from variable nodes to domain nodes.

Figure 2.9 illustrates this modification, with a maximal matching M on the left and its representation with the dummy node on the right.

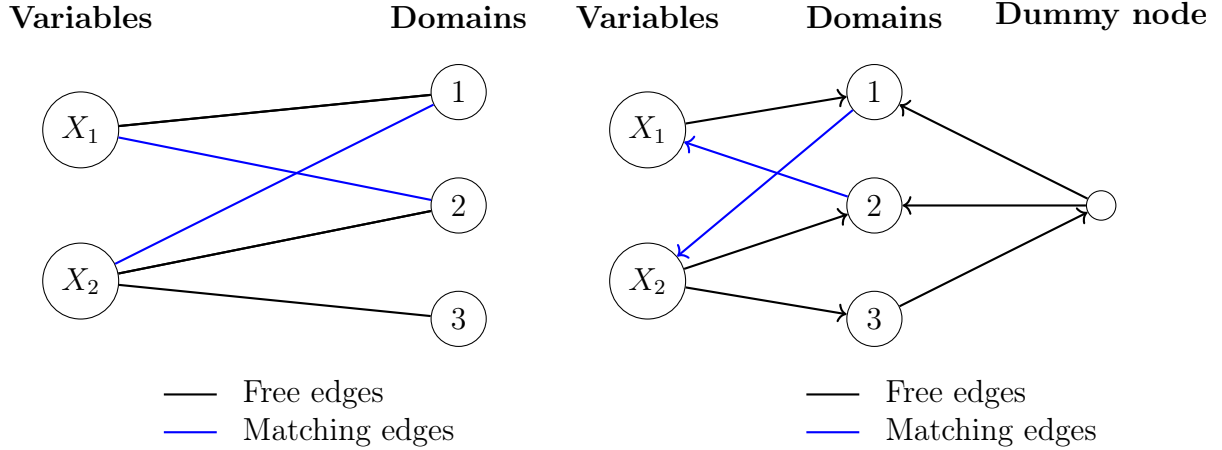


Figure 2.9: Representation of a bipartite graph with a maximal matching of size equal to the number of variables, as well as its representation with edges oriented using a dummy node

In this new representation, an edge belongs to a maximal matching if and only if it either belongs to the initial maximal matching M or is part of a cycle in the transformed graph representing the maximal matching M with the dummy node. In the example above, a cycle $value\ 3 \rightarrow dummy\ node \rightarrow value\ 1 \rightarrow X_2 \rightarrow value\ 3$ exists, which means that the edge connecting X_2 to value 3 belongs to a maximal matching and should not be removed.

To describe the complete algorithm, it is necessary to introduce strongly connected components in a directed graph. A strongly connected component (SCC) is a maximal subset of vertices such that for any u and v in this subset, there exists a path from u to v and from v to u . Based on the modified graph, it is possible to identify the strongly connected components using Kosaraju's algorithm [10,11] or Tarjan's algorithm [12].

Here is the procedure of the complete algorithm. It first finds an arbitrary maximal matching M of size equal to the number of variables. This maximal matching M is initially used to verify feasibility. Next, the algorithm transforms the graph by adding the dummy node and orienting the edges based on the maximal matching M . It then identifies all the strongly connected components in this new directed graph. Finally, it removes all edges (X_i, v_j) if the edge connecting variable X_i to value v_j does not belong to M and if X_i and v_j belong to two different strongly connected components. The removal of an edge between X_i and v_j means that the value v_j is eliminated from the domain of X_i .

Chapter 3

Machine Learning

Our prediction approach in the propagator will rely on machine learning. The following subsections serve to introduce the concepts used in the model.

3.1 Multilayer Perceptron

The multilayer perceptron shown in [Figure 3.1](#) is an assembly of perceptrons arranged in multiple layers, with each layer potentially containing several perceptrons. Its purpose is to predict an output vector $\hat{y}$ based on an input vector X [\[13\]](#).

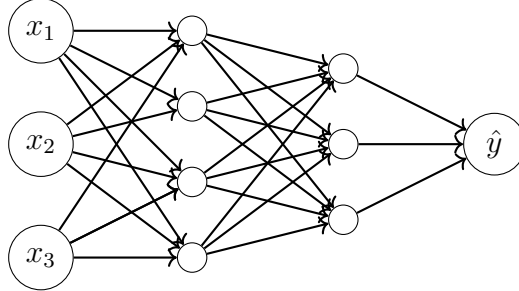


Figure 3.1: Representation of a three-layer MLP with 3 input features and one output

[Equation 3.1](#) and [Equation 3.2](#) show the mathematical relationships between the layers. Each layer i of the neural network performs a linear combination of the inputs from the previous layer $i - 1$ before passing the output through a nonlinear activation function g . This function g allows the network to learn more complex nonlinear functions [\[13\]](#).

$$Z^{[i]} = W^{[i]}A^{[i-1]} + b^{[i]} \quad (3.1)$$

$$A^{[i]} = g^{[i]}(Z^{[i]}) \quad (3.2)$$

[Equation 3.1](#) shows that each layer of the network has a weight matrix $W^{[i]}$ and a bias vector $b^{[i]}$ containing the parameters that the model must learn [\[13,14\]](#). In our model, the activation function g for the hidden layers will be the ReLU function, defined in [Equation 3.3](#).

This function introduces nonlinearity into the model while remaining computationally simple. Moreover, it avoids the vanishing gradient problem, unlike the sigmoid or tanh functions. The output layer will use the sigmoid activation function, defined in [Equation 3.4](#), which produces a value between 0 and 1, as the model's purpose is to predict the probability $\hat{y} = P(y = 1|x)$ [\[13,15\]](#).

$$\text{ReLU}(x) = \begin{cases} x, & \text{if } x > 0, \\ 0, & \text{else.} \end{cases} \quad (3.3)$$

$$\sigma(x) = \frac{1}{1 + e^{-x}} \quad (3.4)$$

To learn the parameters $W^{[l]}$ and $b^{[l]}$ of each layer, the model begins with a forward propagation phase, during which each example from the training set passes through the network to obtain a set of predictions $\hat{y}_i$ and the corresponding true values y_i for each input x_i . A loss function is then used to measure how much the neural network's predictions $\hat{y}_i$ deviate from the expected values y_i . A commonly used loss function for binary classification problems is binary cross-entropy, defined for one example i in [Equation 3.5](#) [\[16\]](#).

$$\text{BCE}(\hat{y}_i, y_i) = -[y_i \log(\hat{y}_i) + (1 - y_i) \log(1 - \hat{y}_i)] \quad (3.5)$$

Next, once all the examples from the training set (or from a batch) have passed through the model, a cost function J is computed over these m examples, as defined in [Equation 3.6](#).

$$J(\theta) = \frac{1}{m} \sum_{i=1}^m \text{Loss}(\hat{y}_i, y_i) \quad (3.6)$$

with θ representing the model parameters (weights and biases).

The goal of the algorithm is to adjust the weights and bias order to minimize this cost function J . To achieve this, a second phase of backpropagation computes the partial derivatives of J with respect to each model parameter. Each parameter w_i or b_i of the model can then be updated using the gradient descent method, as defined in [Equation 3.7](#) and [Equation 3.8](#) [\[13,17\]](#).

$$w_i \leftarrow w_i - \eta \frac{\partial J(\theta)}{\partial w_i} \quad (3.7)$$

$$b_i \leftarrow b_i - \eta \frac{\partial J(\theta)}{\partial b_i} \quad (3.8)$$

Indeed, the function J depends on the various model parameters. As illustrated in [Figure 3.2](#), by computing the partial derivative of a parameter, which represents the slope of the function in the direction of that parameter, it is possible to adjust the parameter in a direction that decreases the value of J .

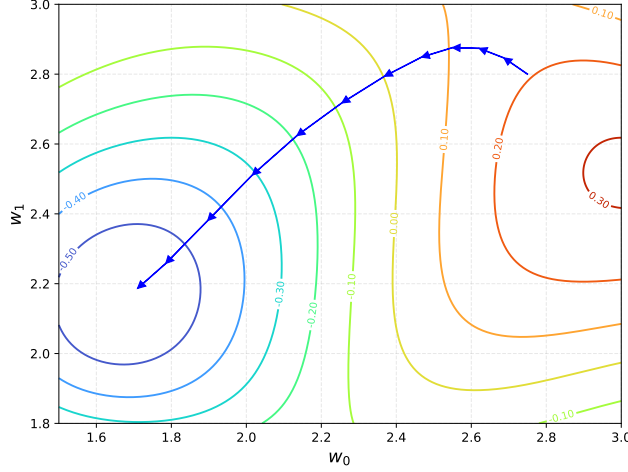


Figure 3.2: Illustration of gradient descent on a function $J(w_0, w_1)$.

The hyperparameter η is the learning rate and determines the step size in the direction of the gradient at each iteration. A step that is too small can slow down convergence, whereas a learning rate that is too large can cause the algorithm to oscillate between values without ever reaching a minimum [17].

During the training phase, a Dropout layer will be used after each activation function. Its purpose is to prevent overfitting by randomly deactivating certain neurons only during training. This helps prevent the network from relying too heavily on a small subset of neurons, thereby reducing overfitting [18].

3.2 Graph Neural Network

Graph Neural Networks (GNNs) are a specific type of neural network designed for data structured as graphs. In such a network, each node v starts with a feature vector $h_v^{[0]}$, which contains various attributes characterizing the node. Then, each node v propagates its vector to all of its neighbors $N(v)$ and aggregates the vectors received from these neighbors. Finally, the vector $h_v^{[l]}$ of node v is updated to a new vector $h_v^{[l+1]}$ through an update function that combines its own vector with the aggregated vectors. Equation 3.9 generally defines this process for one iteration [19].

$$h_v^{[l]} = \text{UPDATE}(h_v^{[l-1]}, \text{AGGREGATE}(h_u^{[l-1]} : u \in N(v))) \quad (3.9)$$

In our model, we will use a specific GNN architecture, the Graph Sample and Aggregate (GraphSAGE), defined in Equation 3.10 [20].

$$h_v^{[l]} = W_1^{[l]} \cdot h_v^{[l-1]} + W_2^{[l]} \cdot \text{mean}_{j \in N(v)} [\sigma(W_3^{[l]} \cdot h_j^{[l-1]} + b^{[l]})] \quad (3.10)$$

The learning of the parameters for each layer is similar to the method used in MLPs. In our implementation, the GraphSAGE layers are directly connected to the MLP model, allowing all parameters to be updated during the same backpropagation phase.

State of the Art

Complete search Search with timeout Objective of the thesis

● Found solution
 ○ Unexplored node
 --- Untraversed edge

21

4.1 Large Neighborhood Search

This first technique is inspired by local search, which balances exploration and exploitation [21]. First, the algorithm starts from a solution. Then, the solver relaxes certain variables by reconstructing their domain while keeping the others fixed: this is the exploration phase. Finally, the solver uses this partially fixed solution to restart a search tree and converge toward another complete solution: this is the exploitation phase. Exploration and exploitation can thus be repeated several times until obtaining a large number of solutions or exceeding a certain stopping criterion when it becomes possible to compare the solutions among themselves [22,23].

Algorithm 1 describes the pseudo-code with $r(x)$ being the function reconstructing a solution from a partial one, $d(x)$ the function selecting and destroying a subset of variables, and $c(x)$ a cost function allowing the comparison of solutions with each other [23].

Algorithm 1: Large Neighborhood Search Algorithm

Input: A feasible solution x
 $x^b \leftarrow x$;
repeat
 $x' \leftarrow r(d(x))$;
 if $\text{accept}(x', x)$ **then**
 $x \leftarrow x'$;
 end
 if $c(x') < c(x^b)$ **then**
 $x^b \leftarrow x'$;
 end
until *stopping criterion is met*;
return x^b ;

Adaptive Large Neighborhood Search (ALNS) is a variant of LNS that adaptively chooses among various solution destruction $d(x)$ and reconstruction $r(x)$ strategies according to their historical effectiveness. Strategies that yield higher-quality solutions are assigned greater weight, increasing their likelihood of being selected in subsequent iterations [23].

A key advantage of this approach is that, by alternating between variable destruction and reconstruction, the algorithm can generate solutions that are uniformly distributed across the classical search tree without requiring a full traversal. Nevertheless, to apply this method, the algorithm must begin with an initial solution, which can be challenging to obtain for certain complex problems.

4.2 Contrastive Predict-and-Search

The concept behind Contrastive Predict-and-Search (ConPaS) involves collecting, across a set of problems, high-quality solutions (optimal or near-optimal) as positive samples and low-quality solutions as negative samples. The negative samples are generated in a targeted

way: either as infeasible solutions produced by randomly perturbing the positive solutions, or as low-quality solutions identified through a maximin optimization, which seeks the worst-performing solutions within the neighborhood of the high-quality ones.

These two sets are employed to train a machine learning model through supervised contrastive learning, enabling it to more effectively differentiate high-quality solutions from low-quality ones.

Once trained, the model assigns scores to the binary variables. ConPaS then greedily selects k_0 variables with the lowest scores (fixed to 0) and k_i variables with the highest scores (fixed to 1), while permitting up to Δ of these fixed variables to be altered during the search. Finally, a classical solver is applied to this reduced problem, subject to neighborhood constraints, to converge to a high-quality solution [24].

A key advantage of this strategy is its ability to identify high-quality solutions without requiring a full traversal of the search tree. A notable drawback (also encountered in the implementation selected for this thesis) is the challenge of generalizing to data drawn from a distribution that differs from that of the training set.

Chapter 5

Propagator Construction

All the code developed to collect the data, build and test the models, construct the propagator, as well as the associated plots and data, can be found in the [GitHub repository \[25\]](#) associated with this project.

5.1 Solver

The solver employed is *MaxiCP*, developed by researchers at the Catholic University of Louvain. This open-source Java solver (MIT license) is publicly available on [GitHub \[26\]](#).

5.2 Data

5.2.1 Instance Selection

To train the machine learning model simulating the propagator, we need to extract the set of variables and their associated domains for the *AllDifferent* constraint both before and after propagation. Indeed, the objective is to predict the updated domains of the variables following propagation.

For this purpose, a subset of instances from the website [XCSP3 \[27\]](#) was downloaded. *XCSP3* is a standardized format for describing CSP instances. It defines the different variables, their domains, and the associated constraints. The official website contains 21,639 instances, reduced to 5,449 instances after filtering for those containing at least one *AllDifferent* constraint.

For the purpose of evaluating our model, we only retain instances that have been fully solved at least once by the original solver. To achieve this, we iterate over the different instance families (Sudoku, NQueen, etc.) in a Round Robin manner and randomly select an instance from the chosen family. The instances are run with a 3-minute timeout. Only instances that complete within the allotted time are kept. The variable selection method during branching is the first-fail technique, which selects at each branch the variable with the smallest domain [28]. After several rounds of the Round Robin, we stop the execution and retain 48 different instances. Some statistics related to the retained instances are summarized in [Table 5.1](#).

Column	Mean	Std. Dev.	Min	Max
# Nodes	206 408.04	484 598.52	0	2 771 998
# Failures	102 108.31	239 149.62	0	1 365 840
# Solutions	1 096.71	3 786.62	0	20 160
ExecutionTime (ms)	113 750.17	391 111.42	3.0	2 334 344.0

Table 5.1: Statistics related to the selected CSP instances

The 48 instances described below contain 17 distinct problem families. These 48 instances are split into two separate folders: the training folder contains 8 distinct problem families, and the test folder contains 9 distinct problem families. The first folder will be used as training data, while the second folder will be used to ensure that the learned model is capable of generalizing its reasoning to problem families it has never encountered during training.

Table 5.2 shows the problem families in each of the two folders, as well as the number of instances for each family.

Train set		Test set	
Family	# Instances	Family	# Instances
All Interval	2	Subisomorphism	3
Allergy	1	Sudoku	8
Alpha	1	Bqwh	7
Kakuro	8	Coloured Queens	2
Knight Tour	1	Costas Array	1
Purdey	1	Open Stacks	1
Queens	2	Langford	2
Schurr Lemma	6	Magic Square	1
		Ortholatin	1

Table 5.2: Number of instances in each family in the train set and in the test set

Appendix Table 11.1 lists the number of nodes, failures, solutions, and calls to the AllDifferent constraint’s propagate function for the training instances. Appendix Table 11.2 presents the same table for the test instances.

5.2.2 Solution Density

For each selected CSP instance, we compute its solution density. Equation 5.1 defines the solution density for a CSP instance, where $|D(x)|$ denotes the domain size of variable x , and X is the set of variables of the problem.

$$\text{Solution Density} = \frac{\#\text{Solutions}}{\prod_{x \in X} |D(x)|} \quad (5.1)$$

This measure allows comparing the intrinsic difficulty of several CSP problems. If the number of possible solutions when all constraints are removed is large, but the number of solutions with the constraints is small, the resulting solution density is low, suggesting a more

difficult problem. Conversely, if the number of possible solutions without the constraints is close to the number of solutions with the constraints, the solution density is high, suggesting an easier problem.

Table 5.3 reports, for the training and test datasets, the solution density of each instance, considering only instances with at least one solution and using a precision of two significant digits.

Training		Test	
Instance	Density	Instance	Density
AllInterval-006	1.6×10^{-7}	ColouredQueens-07	7.8×10^{-38}
AllInterval-009	1.8×10^{-14}	CostasArray-10	4.1×10^{-60}
Allergy	1.5×10^{-5}	MagicSquare-03-sum	2.1×10^{-8}
Alpha	1.6×10^{-37}	Ortholatin-005	5.5×10^{-68}
Kakuro-hard-010-sumdiff	6.3×10^{-80}	Subisomorphism-si2-m4Dr6-m625-04	1.5×10^{-346}
Kakuro-hard-056-sumdiff	3.8×10^{-96}	Subisomorphism-si4-m4Dr2-m625-02	5.1×10^{-698}
Kakuro-hard-062-sumdiff	3.9×10^{-138}	Sudoku-s03b-alldiff	5.1×10^{-78}
Kakuro-hard-137-sumdiff	1.9×10^{-173}	Sudoku-s08c-alldiff	5.1×10^{-78}
Kakuro-medium-035-sumdiff	6.3×10^{-80}	Sudoku-s11b-alldiff	5.1×10^{-78}
Kakuro-medium-062-sumdiff	6.6×10^{-143}	Sudoku-s11c-alldiff	5.1×10^{-78}
Kakuro-medium-086-sumdiff	1.0×10^{-146}	Sudoku-s12a-alldiff	5.1×10^{-78}
Kakuro-medium-129-sumdiff	7.4×10^{-165}	Sudoku-s13a-alldiff	5.1×10^{-78}
KnightTour-06-ext03	9.3×10^{-53}	Sudoku-s13c-alldiff	5.1×10^{-78}
Purdey	1.5×10^{-5}	bqwh-15-106-02_X2	1.6×10^{-56}
Queens-0009-m1	9.1×10^{-7}	bqwh-15-106-07_X2	5.5×10^{-56}
Queens-0012-m1	1.6×10^{-9}	bqwh-15-106-25_X2	1.8×10^{-56}
		bqwh-15-106-51_X2	7.1×10^{-56}
		bqwh-15-106-81_X2	4.9×10^{-56}
		bqwh-18-141-50_X2	1.8×10^{-77}
		bqwh-18-141-75_X2	3.1×10^{-77}
		sudoku368_X2	3.0×10^{-180}

Table 5.3: Solution density of training and test instances (2 significant digits)

It is interesting to note the diversity in solution density. For the training data, we observe a very low solution density for instances belonging to the *Kakuro* family. This is not surprising, given that these instances have a unique solution and a very large search space. For example, *Kakuro-hard-137-sumdiff* has a search space of size 5.22×10^{173} . One can therefore expect it to be more difficult for our solver to find this unique solution. Conversely, although the two instances *Purdey* and *Allergy* also have a unique solution, their search spaces are much smaller, with a size of 65,536 for both. We can therefore expect the solver to find their unique solution more easily. An analogous reasoning can be applied to the test instances.

5.2.3 Data Collection Before and After Propagation

Once these instances have been obtained, the propagator of the *AllDifferent* constraint is modified in order to extract the variables and their domains before and after propagation.

To do so, the *AllDifferent* Domain Consistency constraint is used, because even though it is slower, it provides the best filtering thanks to its domain-consistency property. The input and output data are saved in *JSONL* files. Listing 5.1 shows an example of an entry in these files.

```

1 {
2   "before": {
3     "0": [1, 2],
4     "1": [3],
5     "2": [2, 4],
6     "3": [5],
7     "4": [2, 4]},
8   "after": {
9     "0": [1],
10    "1": [3],
11    "2": [2, 4],
12    "3": [5],
13    "4": [2, 4]
14 }
15 }

```

Listing 5.1: Example of an entry in the JSONL file showing the variables and domains before and after propagation

Once fully executed, the 48 instances described above allow the construction of two databases with 372 947 179 entries for the training data and 98 495 764 entries for the test data.

The figures Figure 5.1, Figure 5.2, Figure 5.3 and Figure 5.4 show, for several training and test instances, a histogram of the domain sizes observed during propagation. The histogram distinguishes domains that have undergone at least one reduction from those that have not.

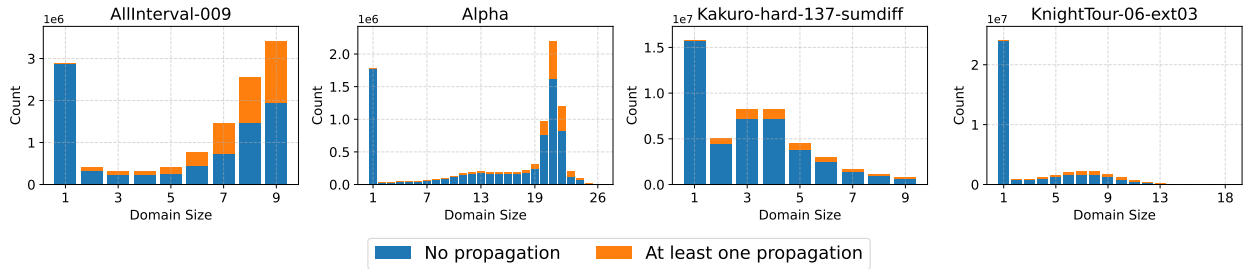


Figure 5.1: Four histograms of training instances illustrating the domain sizes observed during propagation, distinguishing domains that underwent a reduction from those that did not

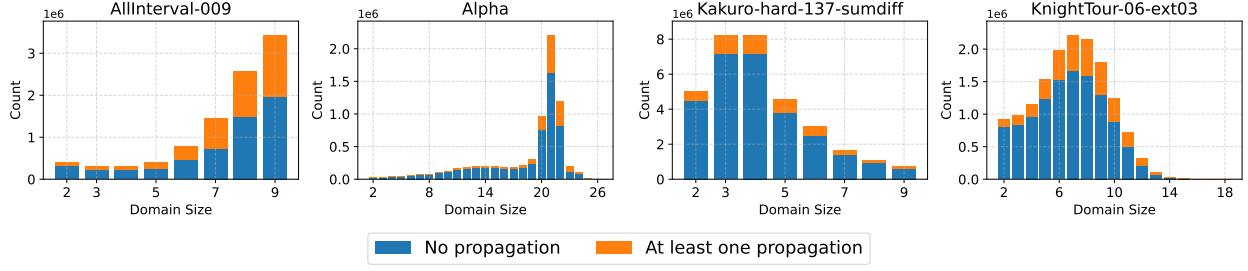


Figure 5.2: Four histograms of training instances illustrating the domain sizes observed during propagation, distinguishing domains that underwent a reduction from those that did not, and removing unit-size domains

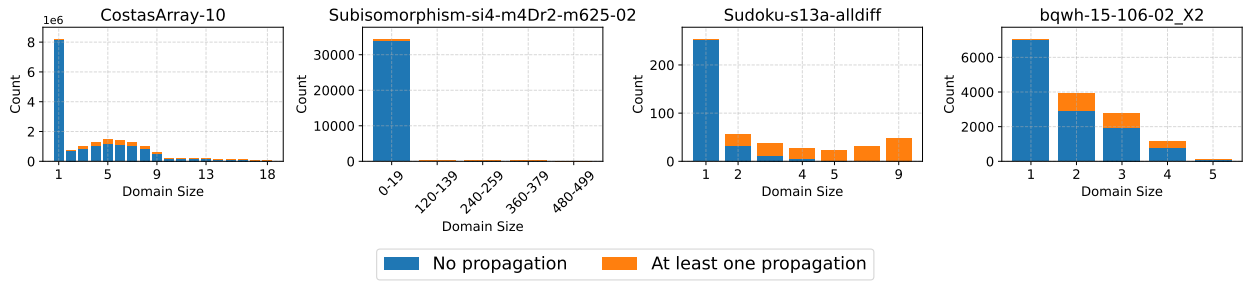


Figure 5.3: Four histograms of test instances illustrating the domain sizes observed during propagation, distinguishing domains that underwent a reduction from those that did not

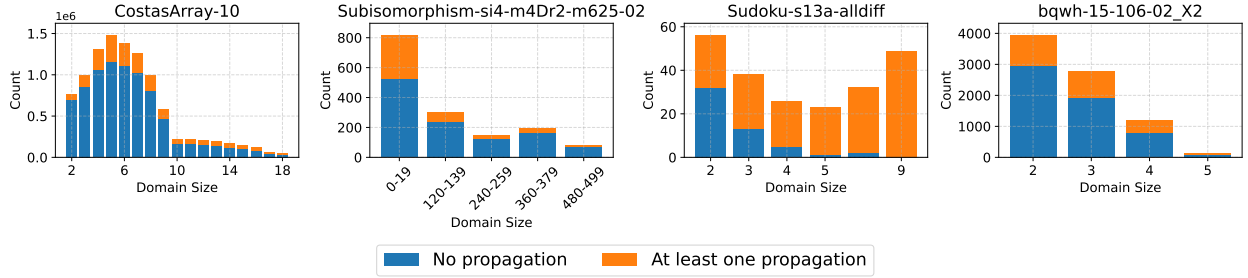


Figure 5.4: Four histograms of test instances illustrating the domain sizes observed during propagation, distinguishing domains that underwent a reduction from those that did not, and removing unit-size domains

In the appendix are [Figure 11.1](#), [Figure 11.2](#), [Figure 11.3](#), and [Figure 11.4](#), which show the same histograms for the entire set of training and test instances, with and without the unit-size domains.

These histograms highlight the diversity of domain sizes observed during propagation for both the training and test data. Moreover, we see that during propagation a large number of domains are already reduced to unit size. This shows that a significant portion of the time is spent at the bottom of the search tree, when many variables are already fixed.

Figure 5.5 and Figure 5.6 below show, for several training and test instances, the number of variables linked by an AllDifferent constraint observed during propagation.

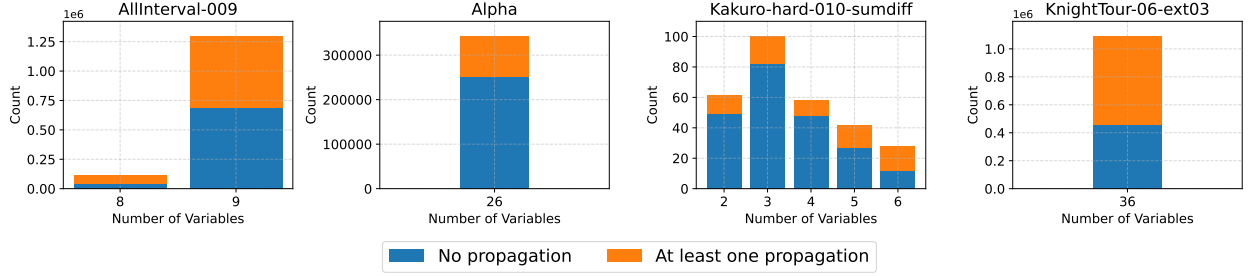


Figure 5.5: Four histograms of the training instances illustrating the number of variables linked by an AllDifferent constraint, observed during propagation, distinguishing between domains that underwent a reduction and those that did not

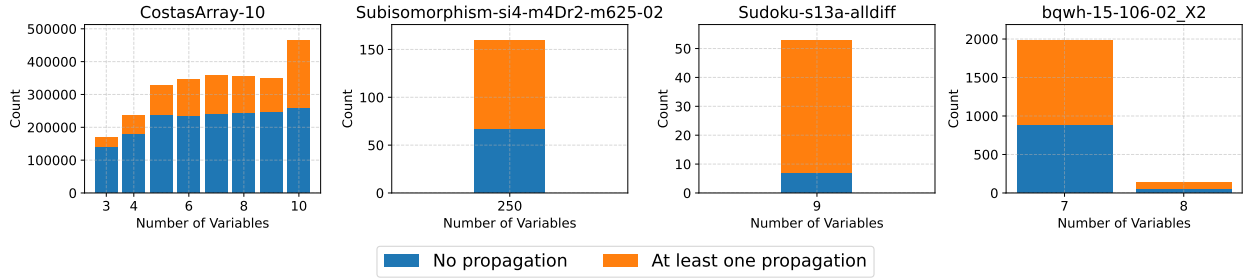


Figure 5.6: Four histograms of the test instances illustrating the number of variables linked by an AllDifferent constraint, observed during propagation, distinguishing between domains that underwent a reduction and those that did not

In the appendix are Figure 11.5 and Figure 11.6, which show the same histograms for the full set of training and test instances, with and without the unit-size domains.

These histograms highlight the large variability in the number of variables linked by an AllDifferent constraint.

Table 5.4 shows the number of entries in the train set JSONL file that underwent at least one domain reduction of a variable during propagation, as well as the number of entries that did not undergo any propagation.

Training data	Number of entries	Proportion (%)
No propagation	321 357 903	86.17
With propagation	51 589 276	13.83
Total	372 947 179	100.0

Table 5.4: Number of propagations that did or did not result in at least one domain reduction in the train set

This difference shows that, in most cases, the propagator is unable to extract any information that would allow a domain reduction. To prevent the model from learning to

systematically remove nothing, a data preprocessing phase is applied. It begins by removing duplicates. Then, the training data are filtered by keeping all entries that exhibit at least one domain reduction, and by randomly sampling 20% of the entries with no reduction.

By applying these filtering rules, the training set is reduced from 372 947 179 *JSON* entries to 4036 191 entries.

5.3 Machine Learning

5.3.1 Data Representation

Before modifying the *AllDifferent* propagator, the first step was to design a machine learning model capable of reproducing the behavior of the *Domain Consistency* propagator using the collected training data.

To do so, we use the data collected before and after propagation to construct a bipartite graph. The nodes on the left represent the variables, while those on the right represent the values present in their respective domains. An edge connects a variable node X_i to a value node v_j if the value v_j belongs to the domain of X_i before propagation.

Each edge is then annotated with a label: this label is 1 if the value v_j is removed from the domain of X_i during propagation, and 0 if it remains.

The goal of our model is to learn from the entire dataset of bipartite graph representations in order to approximate the behavior of the *AllDifferent* DC propagator. Specifically, the model is trained to predict, for each edge in a previously unseen bipartite graph, whether its label should be 1 or 0.

Figure 5.7 presents an illustration of a bipartite graph representing the JSON input shown in Listing 5.1.

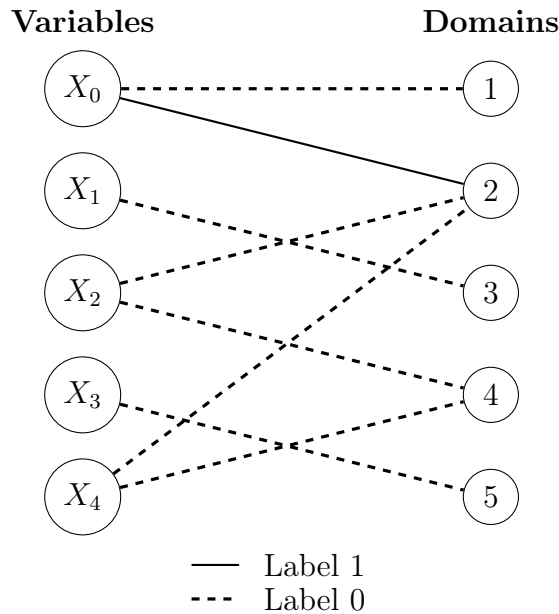


Figure 5.7: Bipartite graph representing the variables and their domains during propagation

5.3.2 Model Architecture

General Architecture

The model was built using Python’s *PyTorch* library, which makes it possible to construct and train machine learning models.

First, the model constructs a vector for each node in the graph. This vector contains, for each node, the normalized degree of the node and its type (0 for variables and 1 for values).

Next, these vectors are passed through a GNN network. The idea is that each node sends its vector to its neighbors, allowing it to combine its own vector with the ones it receives. At the output of the GNN, each node will have an embedding that characterizes the structure in its local neighborhood.

Finally, to label an edge, the two embeddings at the endpoints of the edge are combined and then passed through a multilayer perceptron with a single output, which produces the probability of removing the edge from the graph and therefore removing the value from the domain.

Figure 5.8 illustrates the functioning of the model described above.

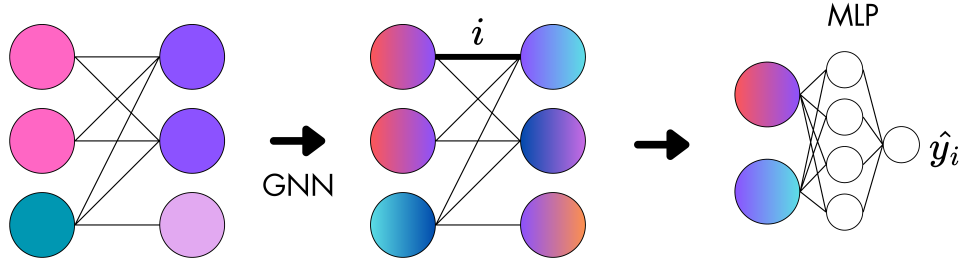


Figure 5.8: Illustration of the edge prediction model pipeline

Loss Function

To train the model effectively, the loss function is essential. Across the different models, the loss function applied to each example is the weighted binary cross-entropy defined in Equation 5.2, where $y_i \in \{0, 1\}$ denotes the ground-truth label of an edge and $\hat{y}_i \in [0, 1]$ is the corresponding model estimate. w_p is a hyperparameter, as described below.

$$\ell(y_i, \hat{y}_i) = -[w_p y_i \log(\hat{y}_i) + (1 - y_i) \log(1 - \hat{y}_i)] \quad (5.2)$$

In our application, two cases are critical: false negatives and false positives. If the model predicts too many false positives, too many values may be removed from the domains, which can lead, in the worst case, to no solution being found. Conversely, if the model predicts too many false negatives, it will keep values that should have been removed, resulting in an excessively large search tree.

In our data, despite the filtering described in the previous sections, a large number of values are kept in the domains during propagation (edges with label 0), leading to a class imbalance. If w_p is set to a unit value, the model may learn to remove no values from the domains, since this would easily yield a good score on the overall loss function. Such a

model would therefore lead to an explosion of false negatives. To address this problem, the hyperparameter w_p can be adjusted to give more or less importance to the positive class. If w_p increases, the model will be more strongly penalized when making mistakes on positive examples, which forces it to learn to recognize them. This will reduce false negatives but will also increase false positives if the model begins to overpredict the positive class on edges labeled 0.

In our case, the parameter w_p is set according to [Equation 5.3](#) [29].

$$w_p = \frac{\text{number of non-removed edges}}{\text{number of removed edges}} \quad (5.3)$$

If the number of removed edges is equal to the number of non-removed edges, the factor w_p becomes unitary and no longer has any impact on the loss function. However, if the number of kept edges is n times larger than the number of removed edges, then $w_p = n$. This therefore adds a factor n to the loss function when a removed edge is inferred in the dataset.

5.4 Integration into the Propagator

5.4.1 Java–Python Communication

In order to use the new model in the propagator of the *AllDifferent* constraint, a new constraint relying exclusively on the prediction model and on a *checker*, described later on, was built in the MaxiCP solver.

Since the MaxiCP code is written in Java while the model is built and used in Python, an API call was set up to manage the connection between the Java propagator and the model running in Python. This solution is not the most optimal, as each API call adds latency and could become a bottleneck if the call time is significant compared to the prediction computation time. However, this solution was kept in order to build a proof of concept.

Another issue that the new constraint may encounter is the need to load the model and its weights at each API call, which would also drastically increase the execution time of a single prediction. To avoid this problem, the Java code launches the Python code, which loads the model only once at the beginning of execution. Then, a socket is created between the Python code and the Java code in order to exchange requests to the model and receive its responses.

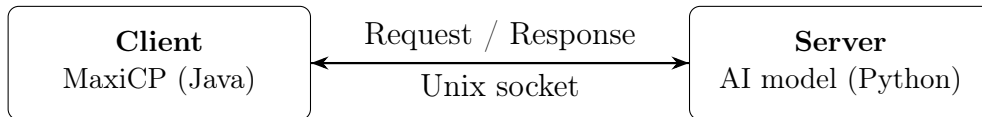


Figure 5.9: Diagram of the client–server architecture between the MaxiCP solver and the AI model through a socket

5.4.2 Checker

The new class representing the *AllDifferent* constraint using the AI model includes a *checker* function that is called at each invocation of the *propagate* function and once in the *post* function. The role of the *checker* function is to remove from the domains of non-fixed variables the values that are already fixed in the domains of assigned variables. This method has several purposes.

First, it provides a simple initial propagation algorithm that reduces the domains. In addition, it helps detect inconsistencies early by removing from a fixed variable its last remaining value if another fixed variable already holds it. Since an empty domain cannot lead to a valid solution, an inconsistency is raised, causing the branch in the search tree to be terminated and backtracked.

For efficiency reasons, the checker uses an array that stores the indices of both fixed and non-fixed variables. The indices of the non-fixed and still active variables are placed on the left side of the array, while the indices of the fixed variables are placed on the right side. A variable stored as a *StateInt* holds an integer that acts as a boundary between the fixed and non-fixed variable indices. Using a *StateInt* allows efficient backtracking: only the integer boundary needs to be restored, without modifying the array of indices itself, which automatically moves the previously fixed variables back to the active side.

Example 5.1. Let us imagine three variables X_1 , X_2 , and X_3 , along with the index table, the domains, and the *StateInt* illustrated in [Figure 5.10](#).

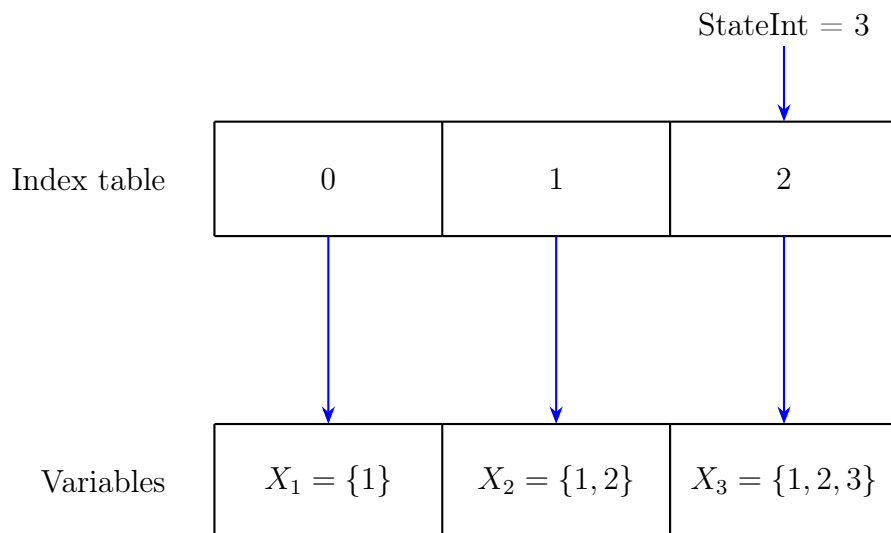


Figure 5.10: Representation of the variables, the index table, and the *StateInt*

The checker iterates through the index array and follows the links to the variables until reaching the *StateInt* (i.e., the end of the index array). During the traversal, it detects that variable X_1 is fixed to 1. It therefore removes the value 1 from the other domains, swaps the index pointing to variable X_1 with the index located at the *StateInt* boundary, and shifts the boundary between fixed and non-fixed variables (by decrementing the *StateInt*). The updated situation is shown in [Figure 5.11](#).

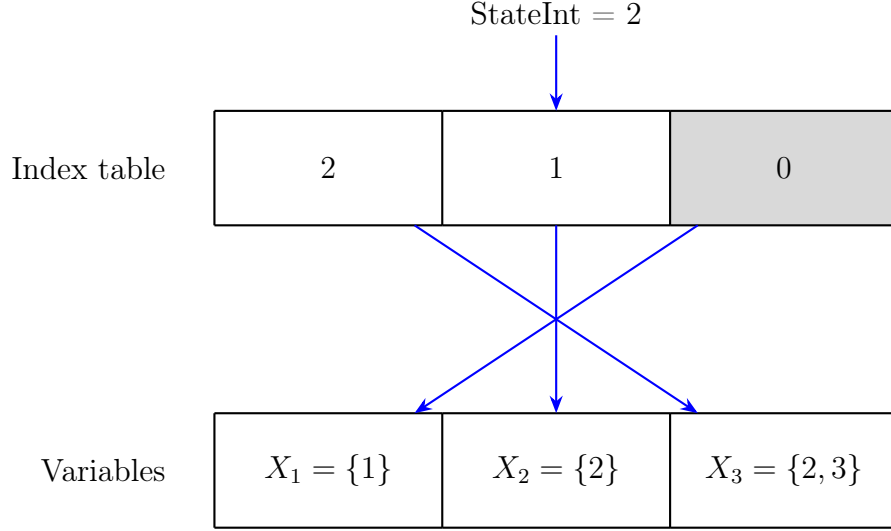


Figure 5.11: Representation of the variables, the index table, and the *StateInt* after one iteration

*This configuration optimizes the checker, as it only traverses the index array from the beginning on the left up to the new *StateInt*. This means it no longer needs to consider index 0, which corresponds to the fixed variable X_1 . In case of a failure, it is sufficient to restore the *StateInt* to its previous state to recover the full set of variables.*

*If we continue the iterations of the checker, it will detect that variable X_2 is fixed to 2. As before, it removes the value 2 from the remaining active domains, swaps the index pointing to X_2 with the index under the *StateInt* (in this case, a swap with itself), and decrements the *StateInt*. The situation is illustrated in [Figure 5.12](#).*

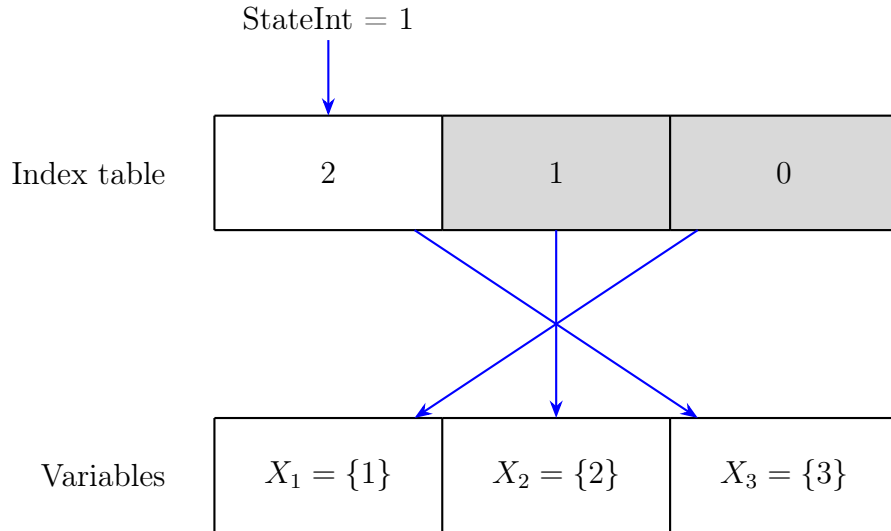


Figure 5.12: Representation of the variables, the index table, and the *StateInt* after two iterations

As before, to restore variable X_2 to the set of active variables, it is enough to restore the

StateInt to its previous value. The checker will once again consider only the indices from the beginning of the index table up to the *StateInt*. Finally, the checker will detect that X_3 is fixed to the value 3. It will therefore swap the index pointing to X_3 with the index under the *StateInt* (again, a swap with itself in this case), remove the value 3 from the domains of the remaining active variables (none in this case), and decrement the *StateInt*. The value of the *StateInt* will then reach 0, meaning that all variables have been fixed.

Chapter 6

Model Creation

To draw conclusions about the quality of the proposed approach, several models with different characteristics were constructed. The sections below describe the various models as well as their specific properties.

6.1 Baseline Model

This first model is considered the baseline model and will be used as a reference when compared with the others. In this model, the vectors constructed at each node contain the following information.

- The normalized degree of the node
- The type of the node (0 for variable nodes and 1 for value nodes)

For each node in the graph, the model projects the input vector of size 2 into a vector of size 16 through a first GNN layer. Then, a second GNN layer projects this vector into another vector of size 16. Next, for each edge to classify, the two embedding vectors of the source and destination nodes are concatenated to obtain a vector of size 32. This 32-dimensional vector is then passed through a two-layer MLP, where each layer reduces the size of the output vector. The vector size is reduced from 32 to 16, and finally to a single scalar value. Finally, the output of the MLP is passed through a sigmoid function in order to obtain a value between 0 and 1.

[Table 6.1](#) summarizes the different layers of the model along with the number of trainable parameters in each layer.

Layer	Parameters
GraphSAGE Layer 1 ($2 \rightarrow 16$)	86
LayerNorm 1	32
GraphSAGE Layer 2 ($16 \rightarrow 16$)	800
LayerNorm 2	32
Edge Prediction MLP	545
Linear ($32 \rightarrow 16$)	528
Linear ($16 \rightarrow 1$)	17
Total	1,495

Table 6.1: Architecture of the baseline model and number of trainable parameters per layer

6.2 Small Model

This model has a reduced size and is used to analyze whether a smaller model can produce good results while accelerating prediction. Although this model is smaller, it is trained on the same dataset as the baseline model.

This model has the same architecture as the baseline model, but the vector sizes have been reduced. For each node, the model input remains a vector of size 2, encoding the degree and the node type. This 2-dimensional vector is then projected through a GNN layer into a vector of size 4, followed by another GNN layer that projects it into another vector of size 4. As in the baseline model, for each edge, the source and destination embeddings are concatenated to obtain a vector of size 8. This vector is passed through two MLP layers that reduce its size from 8 to 4, and finally to a single scalar output. As for the baseline model, the output of the MLP is passed through a sigmoid function in order to obtain a value between 0 and 1.

Table 6.2 summarizes the different layers of the model along with the number of trainable parameters in each layer.

Layer	Parameters
GraphSAGE Layer 1 ($2 \rightarrow 4$)	26
LayerNorm 1	8
GraphSAGE Layer 2 ($4 \rightarrow 4$)	56
LayerNorm 2	8
Edge Prediction MLP	41
Linear ($8 \rightarrow 4$)	36
Linear ($4 \rightarrow 1$)	5
Total	139

Table 6.2: Architecture of the small model and number of trainable parameters per layer

6.3 Random Model Sampled from the Baseline Distribution

This random model does not rely on any machine learning method and is used solely as a reference to compare the performance of the different models studied.

This model attempts to follow the probability distribution given for each edge by the base model. In this regard, the training instances were solved multiple times with different thresholds (from 0 to 0.9 with a step of 0.1) while using the base model. As described in the following section, an edge is removed only when the predicted probability associated with that edge exceeds the threshold. At each call of the *AllDifferent* propagator that relied on the base model, the probabilities predicted for each edge were saved. To avoid generating a huge file containing all the probabilities, a Python dictionary was built to increment a counter for each predicted probability. More concretely, the dictionary contains an entry for every interval of size 0.01 between 0 and 1. For example, the dictionary will have a counter for the interval from 0 to 0.01, another for the interval from 0.01 to 0.02, etc.

Figure 6.1 shows a histogram of the counters for each interval using a logarithmic scale. Figure 6.2 shows the same histogram using a linear scale and removing the interval from 0 to 0.01.

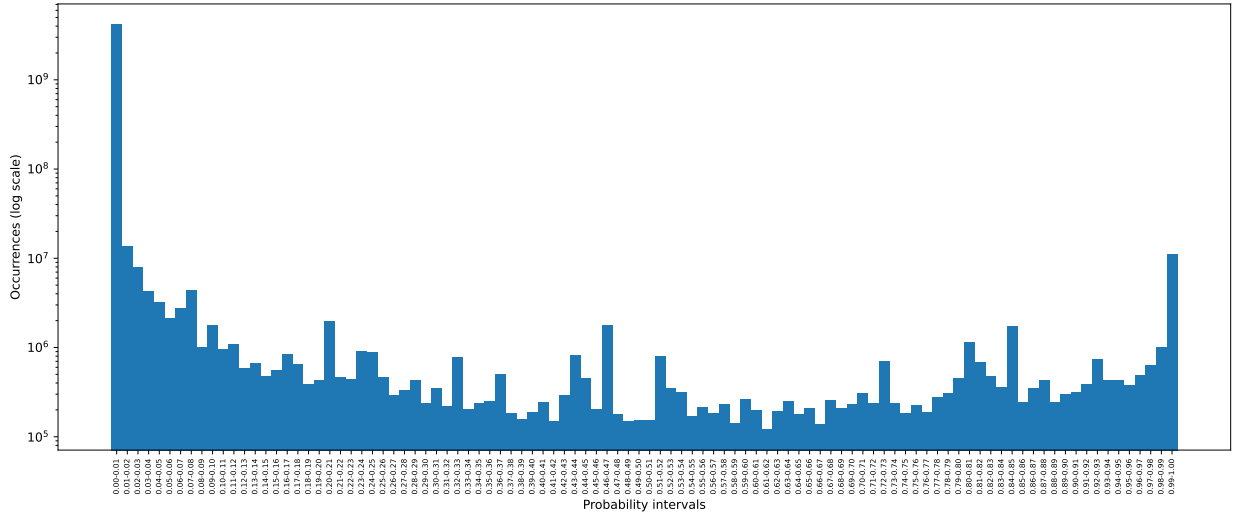


Figure 6.1: Histogram of the occurrences of each predicted probability within an interval of size 0.01 by the base model, using a logarithmic scale

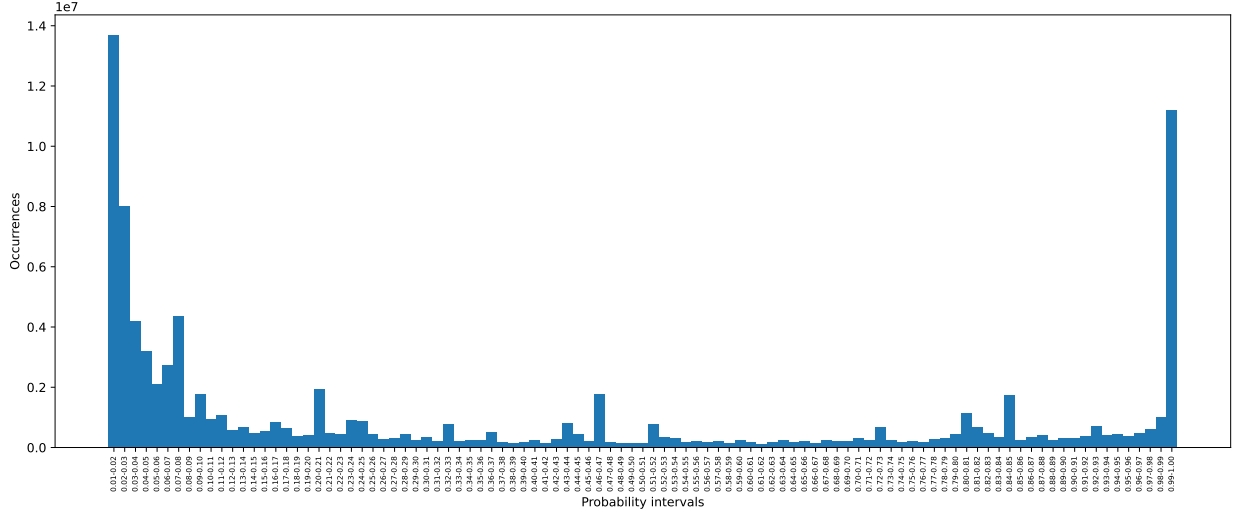


Figure 6.2: Histogram of the occurrences of each predicted probability within an interval of size 0.01 by the base model, using a linear scale and removing the interval from 0 to 0.01

We can clearly see in [Figure 6.1](#) that the vast majority of the probabilities predicted by the base model fall within the interval from 0 to 0.01. Moreover, referring to [Figure 6.2](#), we observe that the histogram follows a U-shaped distribution, with a very large number of occurrences for both small and large probabilities. This suggests that, in most cases, the model is fairly confident in its decisions, predicting low or high probabilities for the edges. Indeed, 99.1994% of the predicted probabilities are either below 0.1 or above 0.9.

The model uses these different counters by randomly selecting an interval, giving more weight to intervals that have a large counter. More concretely, the model maintains a *cumulative list* containing the cumulative sum of the counters of all lower intervals.

[Table 6.3](#) illustrates an example with three intervals between 0 and 0.03. The *Cumulative* column represents the list used by the model.

Interval	Count	Cumulative	Percentage (%)
0.00–0.01	10	10	28.57
0.01–0.02	5	15	14.29
0.02–0.03	20	35	57.14

Table 6.3: Example of intervals, counts, and cumulative values illustrating the weighted sampling method

Based on the maximum value in the *cumulative list*, the model draws a number uniformly between 0 and this maximum value. In the example of [Table 6.3](#), this corresponds to drawing a uniformly random number between 0 and 35. Then, a binary search is used to find the interval corresponding to the drawn number. Interval X is selected if the drawn number falls between the *cumulative* value of the previous interval and the *cumulative* value of interval X . For example, referring to [Table 6.3](#), if the drawn number is 12, the selected interval is

the one from 0.01 to 0.02. Finally, the model draws a uniformly distributed random number within the selected interval to obtain the final predicted probability for the edge.

6.4 Fully Random Model

This final model is the simplest of all. It does not rely on any learning logic or training data. For each edge to be predicted, the model randomly draws a number from a uniform distribution over the interval from 0 to 1. If the drawn value is higher than the threshold, the edge is removed, otherwise the edge is kept.

This fully random model will allow us to test whether the evaluation code for the different models has been correctly implemented. Indeed, in some cases, thanks to the distribution of the training and test data, it is possible to predict the expected behavior of this completely random model, which provides a guarantee regarding the correctness of the code.

Chapter 7

Experiments

This section describes the different experiments conducted to compare the various constructed models with the *Domain Consistency* propagator.

7.1 Evaluation Outside the Propagator

The model is evaluated outside the solver using the data collected during the different propagations of the *Domain Consistency* propagator. To do so, the unfiltered JSON inputs from the training and test instances are used. The evaluation phase reconstructs the graphs linking variables and domains, and the model attempts to predict, for each edge, whether it should be removed (label 1) or kept (label 0). Finally, the predictions are compared to the true labels produced by the *Domain Consistency* propagator, which are also stored in the JSON entries.

7.1.1 Definition of Metrics

Below are the different metrics used to evaluate the models. As explained above, to do so, the predictions made by the models for each edge of the bipartite graph are compared with the values provided by the DC propagator for the same bipartite graph [30,31].

Here are some preliminary notations.

- $TP = \#$ True Positive: number of label 1 predictions when the true label is 1
- $FP = \#$ False Positive: number of label 1 predictions when the true label is 0
- $TN = \#$ True Negative: number of label 0 predictions when the true label is 0
- $FN = \#$ False Negative: number of label 0 predictions when the true label is 1

The first metric is accuracy, defined by [Equation 7.1](#).

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN} \quad (7.1)$$

This first metric provides an overview of the correct predictions (TP and TN) relative to the total number of predictions made. In our application, due to the data distribution, this metric may quickly become uninformative. Indeed, a large portion of the data is labeled 0, indicating that the edge should remain. A model that simply learns not to remove any edge would still achieve a high accuracy, as the number of TN would be high and close to the total number of predictions. It is therefore more relevant to focus on metrics defined for each class.

A first metric defined for each class, is the recall, or for class 0, the specificity. This metric is defined for class 1 in [Equation 7.2](#) and for class 0 in [Equation 7.3](#).

$$\text{Recall}_1 = \frac{TP}{TP + FN} \quad (7.2)$$

$$\text{Specificity} = \text{Recall}_0 = \frac{TN}{TN + FP} \quad (7.3)$$

This metric evaluates, for each class, the number of correct predictions made for that class relative to the number of elements that actually belong to that class. Thanks to this metric, even if the model chooses to classify all edges as belonging to class 0, the number of true positives will be zero, resulting in a recall of zero for class 1. Conversely, if the model classifies all edges as belonging to class 1, the number of true negatives will be zero, leading to a recall of zero for class 0.

Another metric defined for each class is precision. This metric is defined for class 1 in [Equation 7.4](#) and for class 0 in [Equation 7.5](#).

$$\text{Precision}_1 = \frac{TP}{TP + FP} \quad (7.4)$$

$$\text{Precision}_0 = \frac{TN}{TN + FN} \quad (7.5)$$

This metric evaluates, for each class, the number of correct predictions made for that class relative to the total number of elements classified by the model as belonging to that class. This metric makes it possible to determine whether the model frequently predicts an incorrect class. If the number of false positives or false negatives increases, the precision for class 1 or class 0 will decrease accordingly.

Based on the class-wise definitions of recall and precision, we can observe that precision measures the ability of a model to produce correct predictions, whereas recall measures the model's ability to retrieve all elements of that class. The F1 score, defined for both classes in [Equation 7.6](#) and [Equation 7.7](#), combines these two properties into a single metric.

$$F1_1 = 2 \cdot \frac{\text{Precision}_1 \times \text{Recall}_1}{\text{Precision}_1 + \text{Recall}_1} \quad (7.6)$$

$$F1_0 = 2 \cdot \frac{\text{Precision}_0 \times \text{Recall}_0}{\text{Precision}_0 + \text{Recall}_0} \quad (7.7)$$

7.1.2 Practical Evaluation

The tables below summarize the different metrics obtained by running the various models with a fixed threshold of 0.5 on the different training and test instances. As defined in Equation 7.8, an edge i is predicted as belonging to class 1 if the scalar output $z_i^{(L)}$ of the MLP, passed through a sigmoid activation, exceeds the threshold of 0.5. Otherwise, the edge is classified as belonging to class 0.

$$\hat{y}_i = \begin{cases} 1, & \text{if } \sigma(z_i^{(L)}) > 0.5, \\ 0, & \text{else} \end{cases} \quad (7.8)$$

Baseline Model

Appendix Table 11.3 presents the complete evaluation results of the baseline model for the different training instances, and Appendix Table 11.4 for the test instances. Appendix Figure 11.7 shows a histogram of the different metrics for all training instances, and Appendix Figure 11.8 shows the same histogram for all test instances. Table 7.1 summarizes statistics related to the training metrics, and Table 7.2 summarizes the same statistics for the test data. Figure 7.1 presents a boxplot for the different metrics of the training data, and Figure 7.2 presents the same boxplot for the test data.

	Mean	Median	Std	Min	25%	75%	Max
Accuracy (global)	0.993	0.998	0.010	0.970	0.987	1.000	1.000
Precision (class 0)	1.000	1.000	0.001	0.998	0.999	1.000	1.000
Recall (class 0)	0.993	0.999	0.010	0.971	0.987	1.000	1.000
F1-score (class 0)	0.996	0.999	0.005	0.984	0.993	1.000	1.000
Precision (class 1)	0.877	0.985	0.156	0.575	0.743	1.000	1.000
Recall (class 1)	0.988	0.993	0.014	0.951	0.981	1.000	1.000
F1-score (class 1)	0.923	0.984	0.098	0.717	0.847	1.000	1.000

Table 7.1: Statistics of the metrics collected from the training instances when using the baseline model

	Mean	Median	Std	Min	25%	75%	Max
Accuracy (global)	0.911	0.937	0.099	0.612	0.907	0.974	1.000
Precision (class 0)	0.973	0.981	0.042	0.795	0.972	0.994	1.000
Recall (class 0)	0.924	0.952	0.096	0.651	0.926	0.984	1.000
F1-score (class 0)	0.946	0.964	0.066	0.716	0.947	0.985	1.000
Precision (class 1)	0.635	0.660	0.308	0.008	0.515	0.896	1.000
Recall (class 1)	0.807	0.826	0.154	0.484	0.741	0.935	1.000
F1-score (class 1)	0.668	0.721	0.287	0.016	0.606	0.887	1.000

Table 7.2: Statistics of the metrics collected from the test instances when using the baseline model

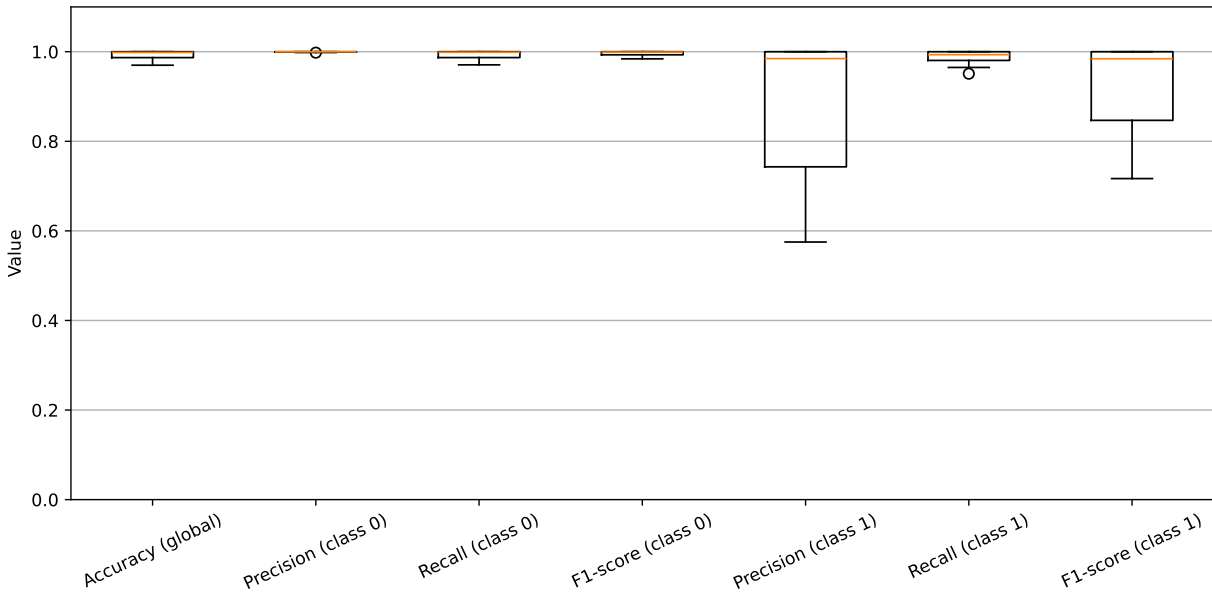


Figure 7.1: Boxplot of the metrics computed from the training data when using the baseline model

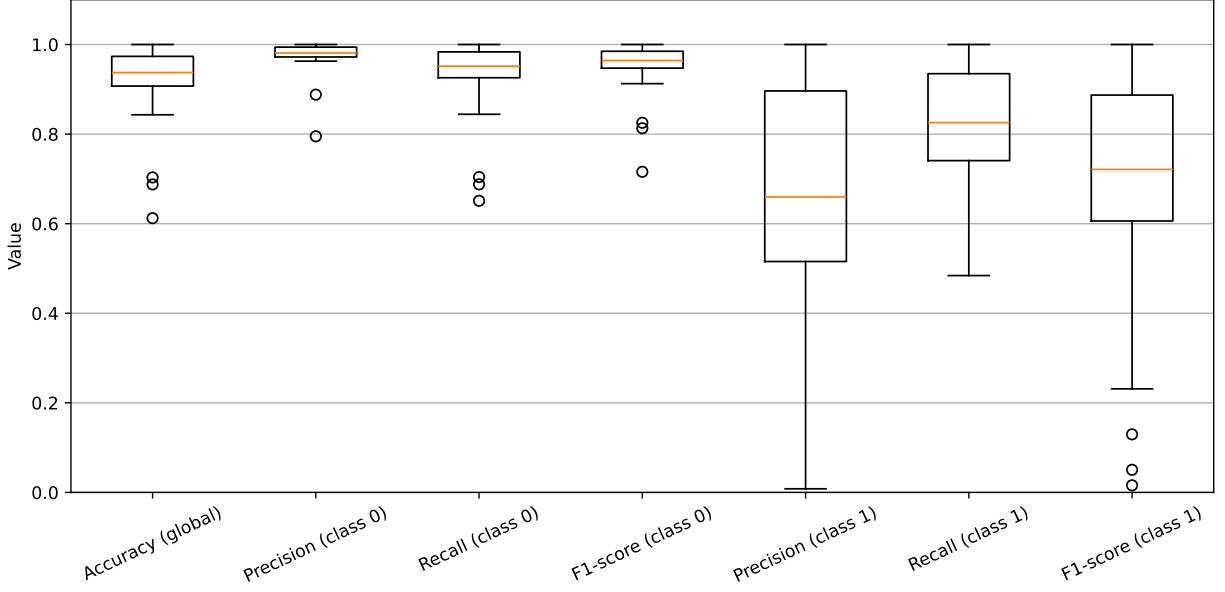


Figure 7.2: Boxplot of the metrics computed from the test data when using the baseline model

We observe that the overall accuracy, as well as the metrics of the majority class, corresponding to keeping a value in a domain (class 0), are relatively high, both with the training data and the unseen test data. For the metrics related to the minority class, corresponding to removing values from the domains (class 1), the recall is relatively good for the training data, with a slight decrease when considering the test data. The precision of the minority class is the poorest metric, both for the training data (average of 87.7%) and for the test data (average of 63.5%). This means that the model predicts a relatively high number of false positives, which results in removing more domain values than the classical solver. We hope that this reduction will allow the search tree to be decreased, leading to a subset of solutions better distributed throughout the tree.

Small Model

As with the baseline model, Appendix Table 11.5 presents all the evaluation metrics obtained when using the small model for each training instance, and Appendix Table 11.6 for the test instances. Appendix Figure 11.9 shows a histogram of the different metrics for the training data when using the small model, and Appendix Figure 11.10 shows the same histogram for the test data. Table 7.3 summarizes the statistics related to the metrics computed with the small model on the training data, and Table 7.4 summarizes the same statistics computed on the test data. Figure 7.3 shows a boxplot of the different metrics computed with the small model on the training data, and Figure 7.4 shows the same boxplot for the test data.

	Mean	Median	Std	Min	25%	75%	Max
Accuracy (global)	0.976	0.987	0.039	0.813	0.972	0.996	0.999
Precision (class 0)	0.983	0.996	0.049	0.769	0.992	0.997	1.000
Recall (class 0)	0.991	0.999	0.013	0.962	0.983	1.000	1.000
F1-score (class 0)	0.986	0.993	0.027	0.870	0.985	0.998	1.000
Precision (class 1)	0.832	0.981	0.205	0.468	0.641	0.997	1.000
Recall (class 1)	0.815	0.904	0.218	0.207	0.808	0.941	1.000
F1-score (class 1)	0.785	0.757	0.178	0.341	0.670	0.966	0.999

Table 7.3: Statistics of the metrics collected on the training instances when using the small model

	Mean	Median	Std	Min	25%	75%	Max
Accuracy (global)	0.928	0.939	0.066	0.712	0.930	0.958	1.000
Precision (class 0)	0.946	0.939	0.045	0.807	0.928	0.985	1.000
Recall (class 0)	0.975	0.999	0.064	0.712	0.993	1.000	1.000
F1-score (class 0)	0.958	0.966	0.041	0.824	0.959	0.977	1.000
Precision (class 1)	0.809	0.988	0.332	0.031	0.780	1.000	1.000
Recall (class 1)	0.566	0.615	0.225	0.150	0.369	0.718	1.000
F1-score (class 1)	0.587	0.546	0.260	0.060	0.477	0.805	1.000

Table 7.4: Statistics of the metrics collected on the test instances when using the small model

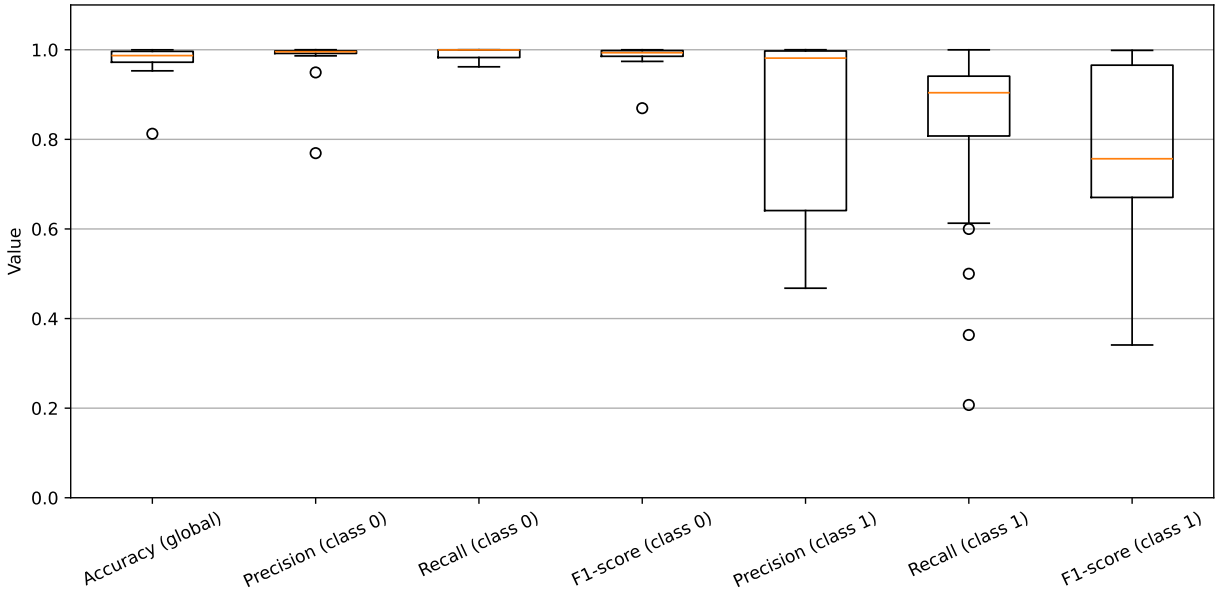


Figure 7.3: Boxplot of the metrics computed from the training data when using the small model

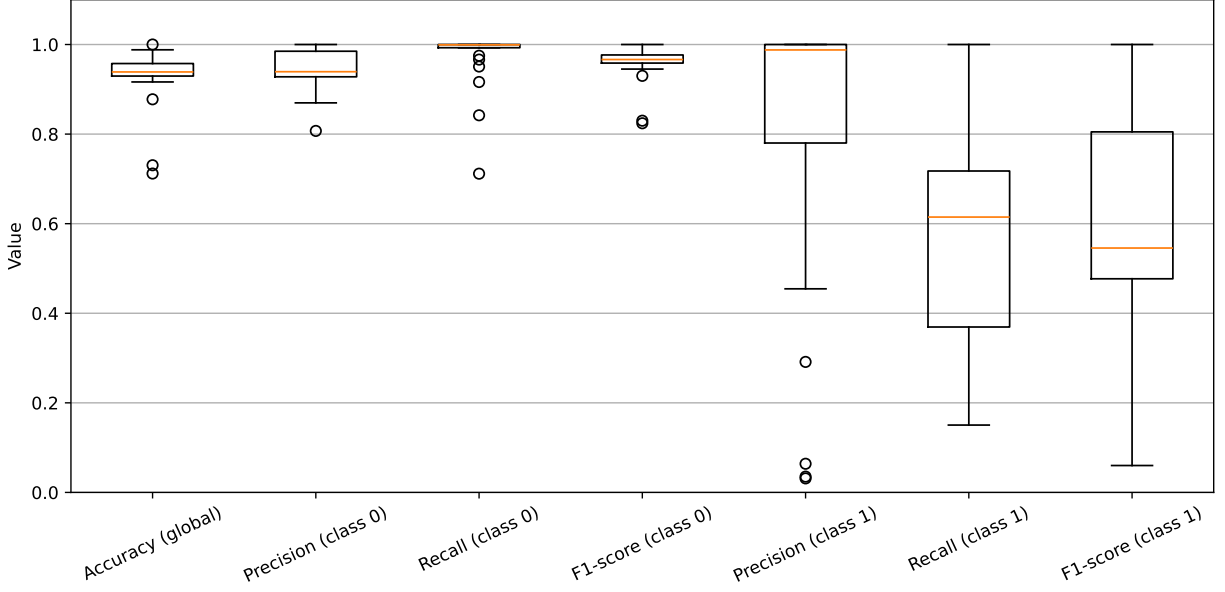


Figure 7.4: Boxplot of the metrics computed from the test data when using the small model

We observe that the metrics for the majority class (class 0) are relatively high for both the training and test sets. This means that the model successfully identifies the edges that should be kept. For the minority class (class 1), the average precision also remains quite high for the training set (83.2%) and the test set (80.9%). This indicates that when the model predicts this minority class, it is rarely wrong. The average recall for the same minority class is also relatively high on the training set (81.5%) but drops significantly on the test set (56.6%). This means that the model often classifies an edge that should be removed as one that should be kept. We therefore expect that, on data never seen during training, we will obtain a large search tree but without removing a significant number of potential solutions. Indeed, when the model keeps an edge that should have been removed, it takes no risk and reduces the domains less, which increases the size of the search without excessively pruning branches that may contain solutions.

When comparing the large model with the small model, we observe, thanks to [Figure 7.1](#) and [Figure 7.3](#), that the large model provides better overall performance than the small one on the training data. Concerning the test data and the minority class, as shown in [Figure 7.2](#) and [Figure 7.4](#), the large model offers on average a better recall than the small model, while the small model offers better precision than the large one.

Random Model Sampled from the Baseline Distribution

As with the other models, [Appendix Table 11.7](#) presents all evaluation metrics for the random model following the base model’s distribution for each training instance, and [Table 11.8](#) for the test instances. [Appendix Figure 11.11](#) shows a histogram of the different measures for the training data with this random model, and [Appendix Figure 11.12](#) for the test data. [Table 7.5](#) provides statistics related to the metrics computed with this random model on the training data, and [Table 7.6](#) presents the same statistics computed on the test data. [Figure 7.5](#) shows

a box plot for the different metrics computed with this random model on the training data, and [Figure 7.6](#) shows the same box plot on the test data.

	Mean	Median	Std	Min	25%	75%	Max
Accuracy (global)	0.935	0.955	0.073	0.625	0.941	0.961	0.981
Precision (class 0)	0.941	0.961	0.075	0.625	0.947	0.968	0.988
Recall (class 0)	0.993	0.993	0.002	0.987	0.993	0.993	1.000
F1-score (class 0)	0.965	0.977	0.046	0.769	0.969	0.980	0.991
Precision (class 1)	0.038	0.034	0.027	0.000	0.021	0.050	0.122
Recall (class 1)	0.006	0.007	0.003	0.000	0.007	0.007	0.009
F1-score (class 1)	0.010	0.012	0.005	0.000	0.009	0.012	0.018

Table 7.5: Statistics of the metrics collected on the training instances when using the random model following the base model’s distribution

	Mean	Median	Std	Min	25%	75%	Max
Accuracy (global)	0.880	0.896	0.082	0.703	0.818	0.947	0.989
Precision (class 0)	0.886	0.902	0.084	0.706	0.822	0.954	0.995
Recall (class 0)	0.993	0.993	0.001	0.988	0.993	0.993	0.995
F1-score (class 0)	0.934	0.945	0.048	0.825	0.900	0.973	0.994
Precision (class 1)	0.100	0.077	0.094	0.000	0.043	0.106	0.333
Recall (class 1)	0.006	0.007	0.002	0.000	0.006	0.007	0.009
F1-score (class 1)	0.011	0.012	0.005	0.000	0.007	0.014	0.018

Table 7.6: Statistics of the metrics collected on the test instances when using the random model following the base model’s distribution

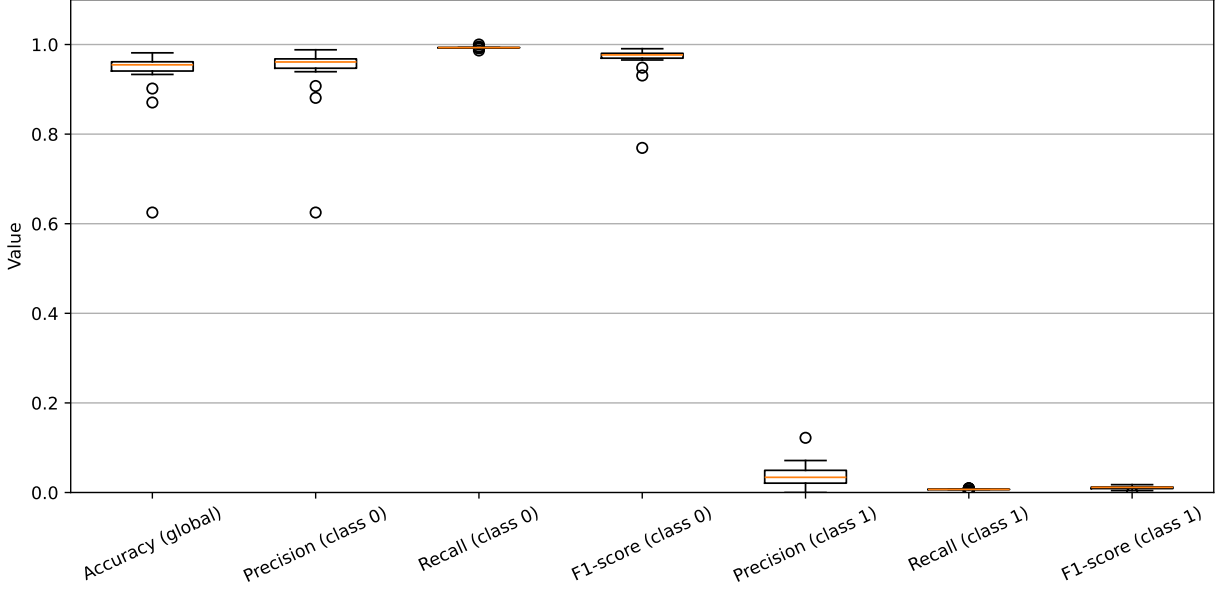


Figure 7.5: Box plot of the metrics computed on the training data when using the random model following the base model’s distribution

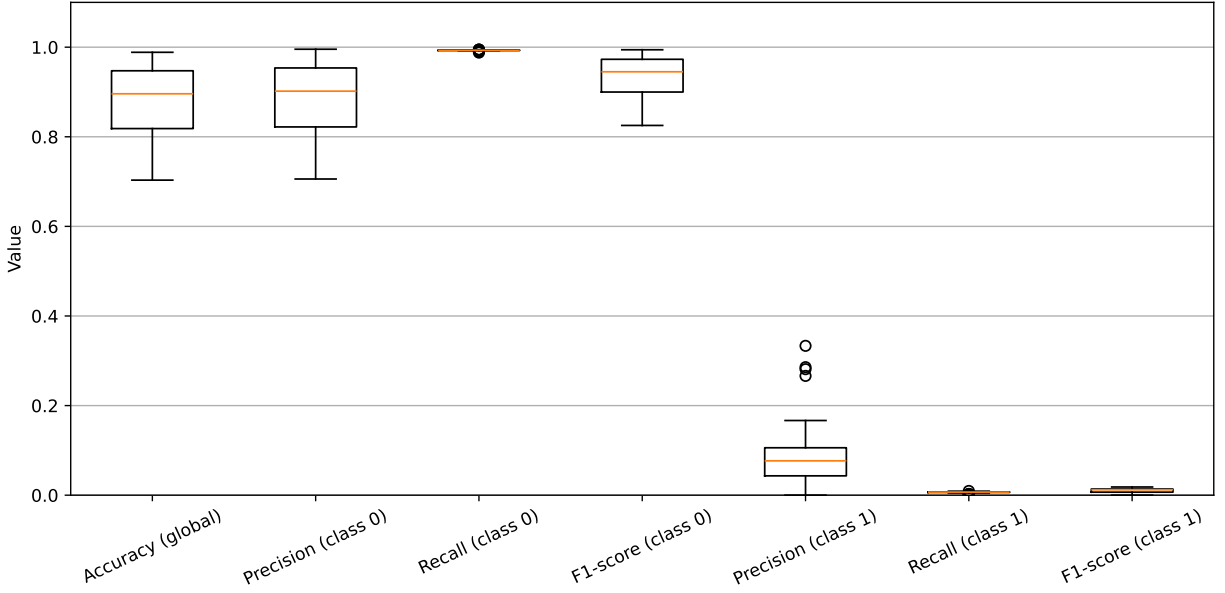


Figure 7.6: Box plot of the metrics computed on the test data when using the random model following the base model’s distribution

Although we obtain good metrics for the majority class corresponding to the edges to keep, we observe a collapse in the metrics for the minority class corresponding to the edges to remove. The same behavior occurs both on the training data and on the test data. Referring to [Figure 6.1](#), we notice that most predictions from the base model correspond to the majority

class. Moreover, as shown in [Table 5.4](#) for the training data, a large majority of edges should not be removed. Therefore, the model will predict in most cases that the edge should not be removed and will be correct most of the time, which increases the precision and recall for the majority class. Referring to [Equation 7.3](#) and [Equation 7.5](#), we can deduce that the number of True Negatives (TN) will be high compared to the number of False Positives (FP) and False Negatives (FN). The opposite effect occurs for the minority class, causing the recall, precision, and F1 score for this class to drop.

Fully Random Model

Since this model did not benefit from any training, the training and test datasets were merged into a single dataset in order to obtain a single evaluation on the data. Indeed, as explained earlier, for each edge this model simply draws a random number between 0 and 1 from a uniform distribution and removes the edge (class 1) if the drawn number is greater than the threshold, and keeps it (class 0) otherwise.

Appendix [Table 11.9](#) presents all the evaluation results of the fully random model for the different training and test instances. Appendix [Figure 11.13](#) shows a histogram of the different metrics for all training and test instances. [Table 7.7](#) provides statistics related to the training and test metrics. [Figure 7.7](#) displays a box plot for the different metrics of the training and test data.

	Mean	Median	Std	Min	25%	75%	Max
Accuracy (global)	0.504	0.500	0.028	0.459	0.500	0.502	0.688
Precision (class 0)	0.914	0.948	0.075	0.703	0.898	0.967	0.995
Recall (class 0)	0.505	0.500	0.037	0.467	0.500	0.501	0.750
F1-score (class 0)	0.648	0.654	0.025	0.588	0.643	0.660	0.750
Precision (class 1)	0.094	0.052	0.103	0.005	0.036	0.104	0.583
Recall (class 1)	0.506	0.500	0.028	0.389	0.499	0.508	0.600
F1-score (class 1)	0.141	0.094	0.115	0.009	0.067	0.173	0.583

Table 7.7: Descriptive statistics of the metrics collected on the training and test instances when using the fully random model

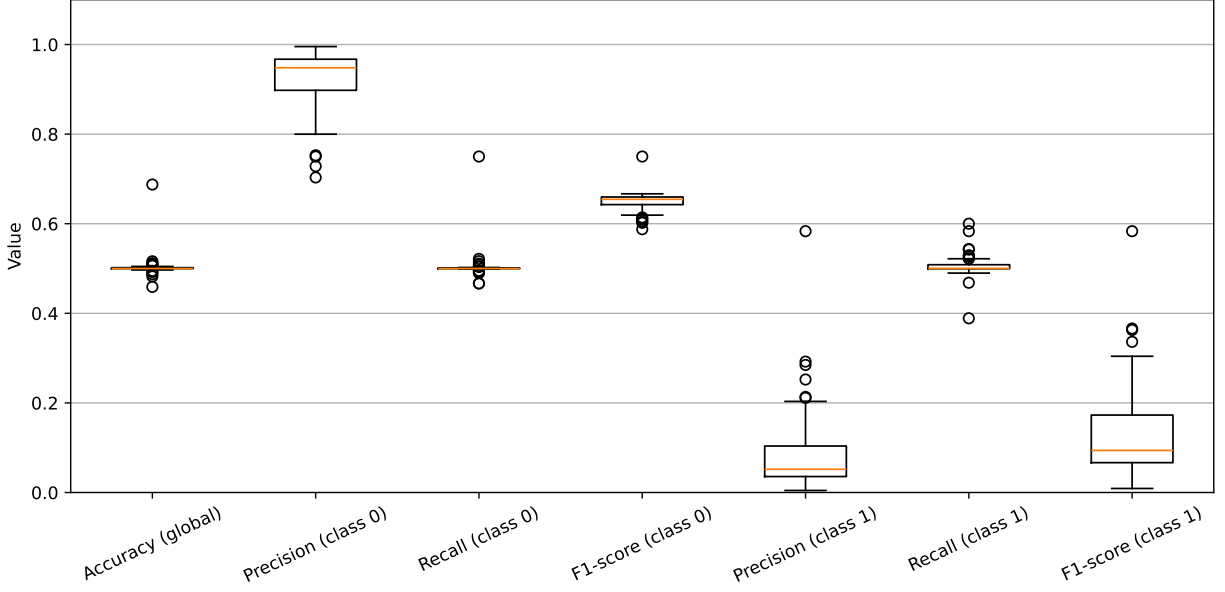


Figure 7.7: Box plot of the metrics computed on the training and test data when using the fully random model

As expected with a completely random model following a uniform distribution, the average overall accuracy is close to 50%. We observe that the average precision for the majority class is high (91.4%) and very low for the minority class (9.4%). This behavior is expected for this kind of problem with unbalanced data and a uniformly random model with a threshold of 0.5. Indeed, referring to Equation 7.4 and Equation 7.5, the model will obtain many True Negatives (TN) and False Positives (FP) but few True Positives (TP) and False Negatives (FN), which increases the precision of the majority class and decreases the precision of the minority class. The recall for both the majority and minority classes is close to 50%, which is expected. Referring to Equation 7.2 and Equation 7.3, the number of True Positives (TP) and False Negatives (FN) is low and similar in value, whereas the number of True Negatives (TN) and False Positives (FP) is high and also close in value.

Apart from these metrics, as previously explained, the random models will be used to compare, for different thresholds, the number of solutions found and the number of nodes explored between a trained model and a completely random model.

7.2 Number of Solutions

This section shows the number of solutions found when using the new propagators, for different instances and different thresholds θ , compared to the classical Domain Consistency propagator. The variable threshold θ makes it possible to specify the model's sensitivity in predicting whether a value should be removed from a domain. Indeed, the model's output behaves according to Equation 7.9. An edge i is predicted as belonging to class 1 if the scalar output $z_i^{(L)}$ of the MLP, passed through a sigmoid activation, exceeds the threshold θ . Otherwise, the edge is classified as belonging to class 0.

$$\hat{y}_i = \begin{cases} 1, & \text{if } \sigma(z_i^{(L)}) > \theta, \\ 0, & \text{else} \end{cases} \quad (7.9)$$

For this section, unlike during training data collection and the previous evaluation section, only the instances that have at least one solution in the classical solver will be used. Indeed, it would not be relevant to evaluate the number of solutions found using the machine learning model on instances that have no solution.

7.2.1 Baseline Model

The graphs showing, for the training instances, the number of solutions, nodes, and failures computed with the base model for different thresholds are found in [Figure 11.14](#), [Figure 11.15](#), and [Figure 11.16](#), respectively. The same graphs obtained on the test instances are found in Appendices [Figure 11.17](#), [Figure 11.18](#), and [Figure 11.19](#). Below are some graphs that are interesting to interpret.

[Figure 7.8](#) compares the number of nodes and solutions found as a function of the threshold for the instance *KnightTour-06-ext03*, which is part of the training instances. The red line corresponds to the values obtained when running the instance with the classical *AllDifferent* propagator without machine learning.

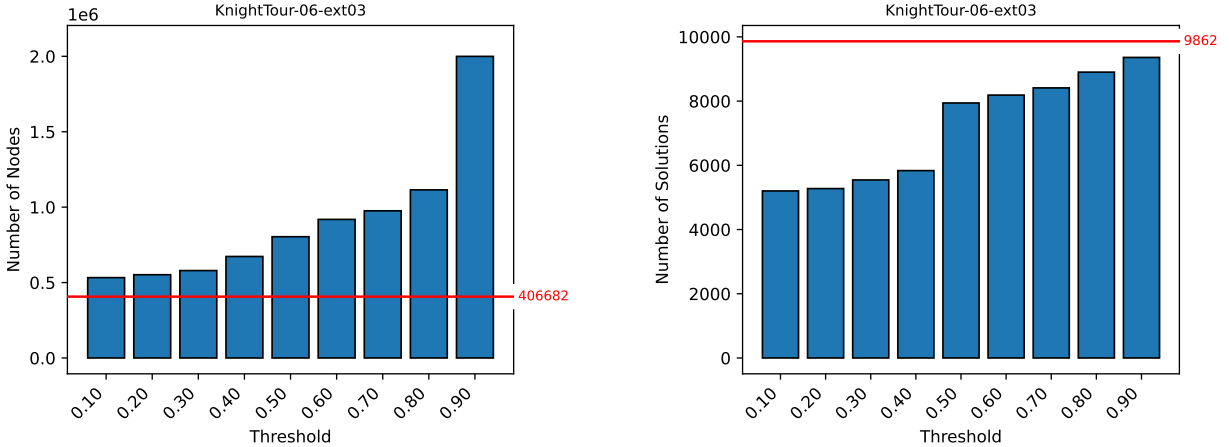


Figure 7.8: Comparison of the number of nodes (left) and solutions (right) as a function of the threshold for the instance *KnightTour-06-ext03*

What is interesting to observe is that it is possible to vary the number of solutions and the number of nodes explored by changing the threshold of the machine learning model. Indeed, when setting a low threshold, the probabilities predicted by the model for each edge are more likely to be above this threshold, leading to the removal of the edge and therefore the domain value. Conversely, when setting a high threshold, more probabilities fall below this threshold, which results in keeping the edge and thus preserving the domain value. What is disappointing for this particular instance is that, for every threshold, the number of nodes explored is higher than the number of nodes explored with the classical propagator.

Figure 7.9 highlights the same behavior for the *Ortholatin-005* instance, which is part of the test instances.

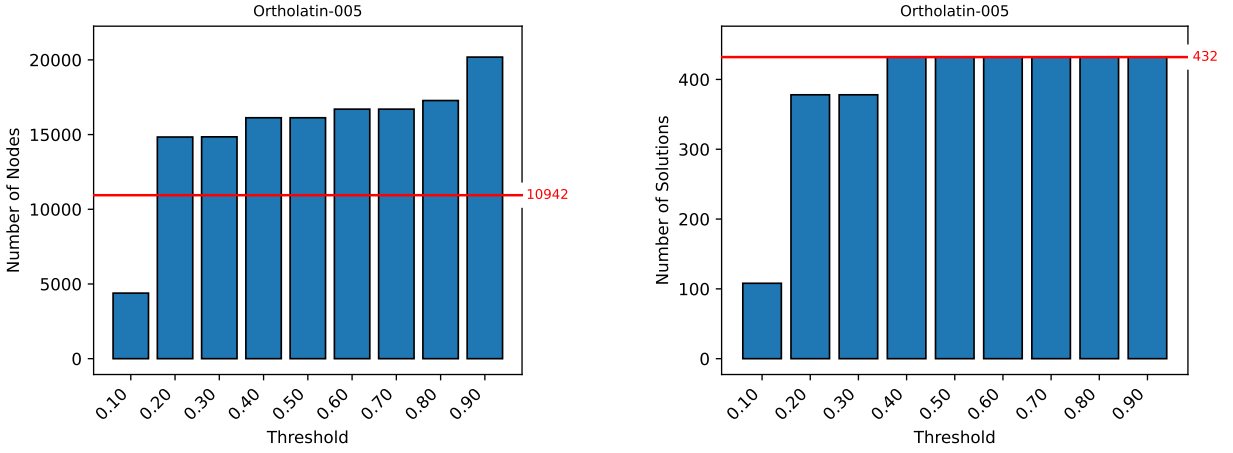


Figure 7.9: Comparison of the number of nodes (left) and solutions (right) as a function of the threshold for the *Ortholatin-005* instance

Figure 7.10 and Figure 7.11 show the same comparison between the number of nodes and solutions for different thresholds for the *CostasArray-10* and *MagicSquare-03-sum* instances, two test instances that were never seen by the model.

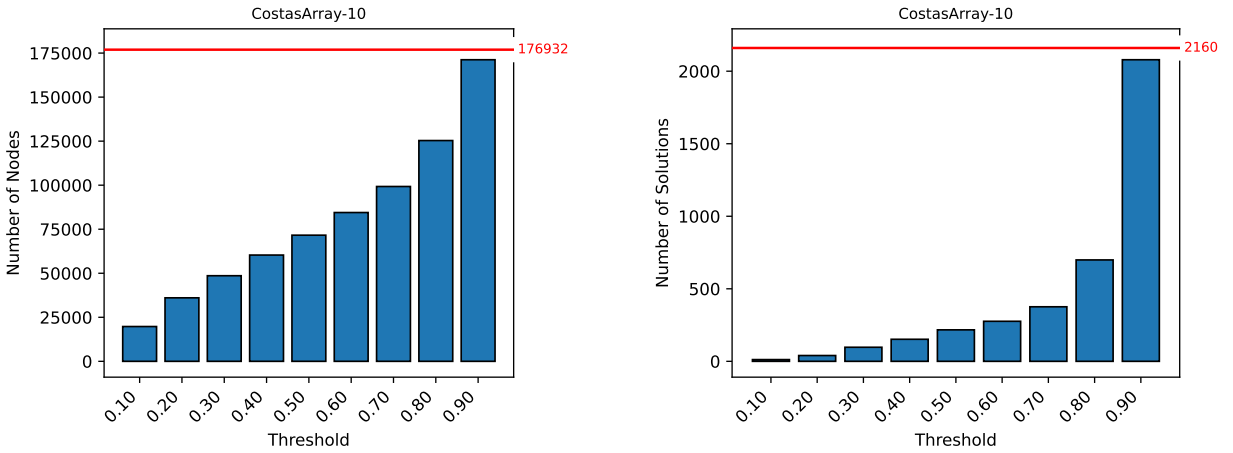


Figure 7.10: Comparison of the number of nodes (left) and solutions (right) as a function of the threshold for the *CostasArray-10* instance

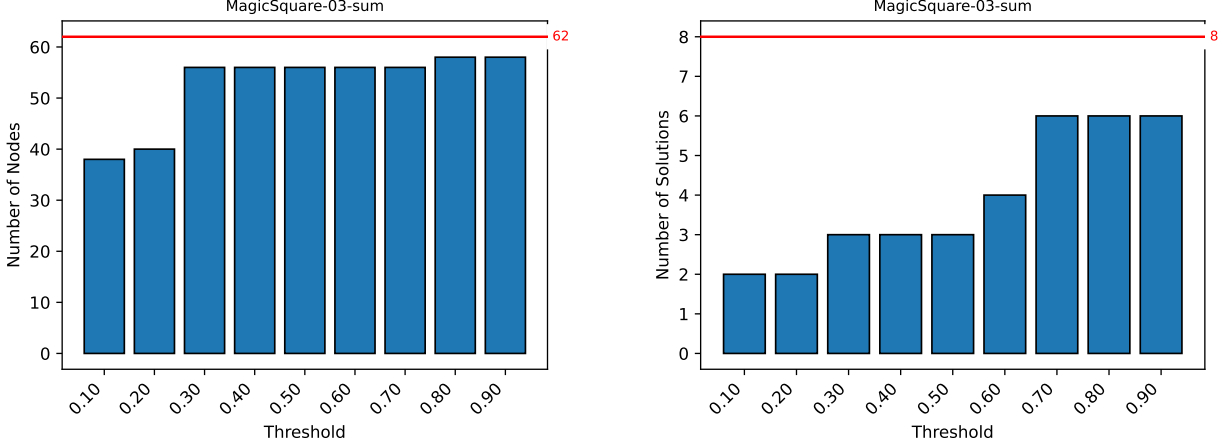


Figure 7.11: Comparison of the number of nodes (left) and solutions (right) as a function of the threshold for the *MagicSquare-03-sum* instance

These two instances are very interesting because they illustrate the possibility of obtaining, by varying the threshold, a subset of solutions while exploring fewer nodes than the classical solver without machine learning. Moreover, the almost linear profile of the number of nodes explored as a function of the threshold shown in Figure 7.10 corresponds to the desired behavior.

It is important to note that although this evolution profile as a function of the threshold is not obtained for all instances, the solver succeeds, for some of them, in finding solutions while exploring fewer nodes than the classical solver. For example, as shown in Figure 7.12, the solver managed, with a threshold of 0.9, to find the unique solution of the *Kakuro-hard-010-sumdif* instance while exploring fewer nodes than the classical solver without machine learning.

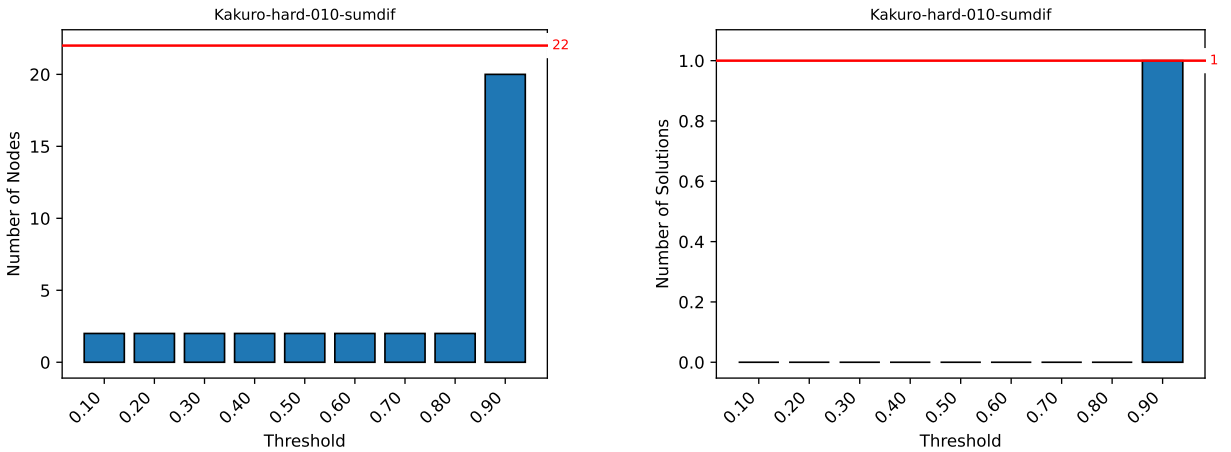


Figure 7.12: Comparison of the number of nodes (left) and solutions (right) as a function of the threshold for the *Kakuro-hard-010-sumdif* instance

7.2.2 Small Model

As with the base model, the graphs of the number of solutions, nodes, and failures for the training instances computed with the small model are found in Appendix Figure 11.20, Figure 11.21, and Figure 11.22. The same graphs obtained on the test instances are found in Appendix Figure 11.23, Figure 11.24, and Figure 11.25.

First, by comparing Appendix Figure 11.18 and Appendix Figure 11.24, we can notice that, in general, for the test instances, the small model explores a larger number of nodes than the base model. On the other hand, based on Figure 11.17 and Figure 11.23, the small model generally succeeds in finding a greater number of solutions than the base model. Figure 7.13 illustrates an example by comparing the number of nodes and solutions for the *CostasArray-10* instance when using the base model and the small model.

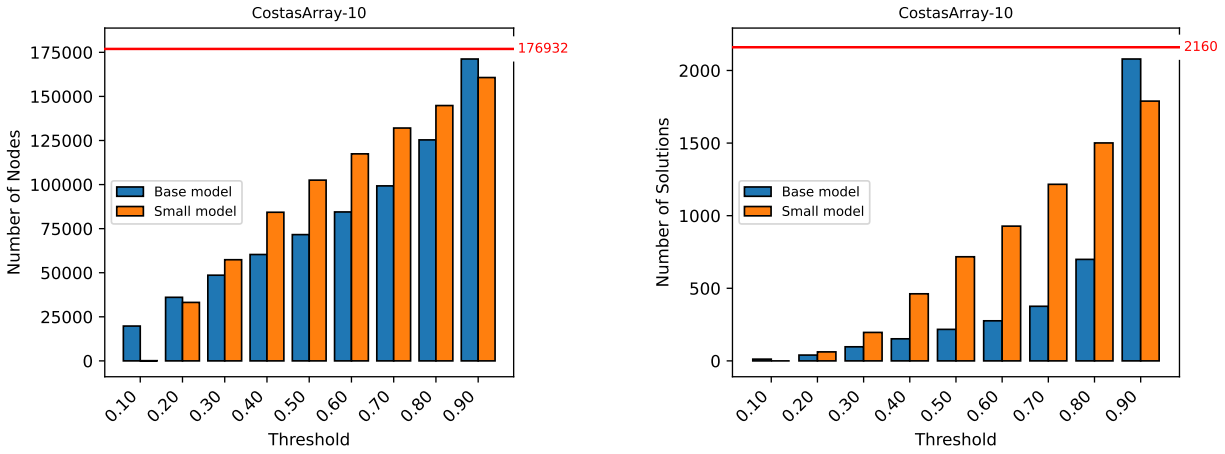


Figure 7.13: Comparison of the number of nodes (left) and solutions (right) as a function of the threshold for the *CostasArray-10* instance, comparing the base model with the small model

This outcome was expected. Indeed, based on the box plots in Figure 7.2 and Figure 7.4, we can see that the average recall for the class representing the edges to remove is lower for the small model. This means that, for the small model, the number of False Negatives is high and that many edges belonging to the class of edges to remove will be kept. The direct impact of the small model's decision to be more conservative when removing an edge is an increase in the search tree and therefore in the number of nodes, but also in the number of solutions found.

Regarding the training instances, as illustrated in Appendix Figure 11.14 and Appendix Figure 11.20, we can see that for several problems with a single solution, the base model is able to find it with a threshold of 0.9, whereas the small model does not.

Finally, as shown in Appendices Figure 11.14, Figure 11.15, Figure 11.20, and Figure 11.21, it is interesting to note that for six instances belonging to the training data (AllInterval-006, AllInterval-009, Allergy, Purdey, Queens-0009-m1, and Queens-0012-m1), the base model and the small model obtain, for all thresholds, the same number of nodes and solutions as those obtained when using the propagator without machine learning. This suggests either that the

AllDifferent constraint propagator is rarely, if ever, called for these instances, or that the propagation is trivial and can be fully learned without error by the models. As shown in Table 7.8, for four of the six instances, the number of calls to the propagate function is quite significant, which supports the second hypothesis for these four instances.

Instance	Model	0.10	0.20	0.30	0.40	0.50	0.60	0.70	0.80	0.90
<i>AllInterval-006</i>	Base	2694	2699	2699	2703	2704	2700	2700	2700	2709
	Small	2648	2703	2709	2709	2709	2709	2709	2709	2709
<i>AllInterval-009</i>	Base	1435695	1440779	1441593	1442666	1443476	1446789	1446750	1448528	1450642
	Small	1425974	1441713	1443065	1450084	1450910	1451058	1451558	1451453	1451594
<i>Allergy</i>	Base	3	3	3	3	3	3	3	3	3
	Small	3	3	3	3	3	3	3	3	3
<i>Purdey</i>	Base	8	8	8	8	8	8	8	8	8
	Small	8	8	8	8	8	8	8	8	8
<i>Queens-0009-m1</i>	Base	29724	29752	29762	29760	29769	29722	29722	29726	29724
	Small	29592	29786	29797	29785	29825	29826	29807	29807	29939
<i>Queens-0012-m1</i>	Base	4399889	4408130	4410073	4411789	4415794	4411762	4413015	4416451	4416110
	Small	4370994	4416932	4422159	4423153	4426214	4427179	4425362	4427389	4451363

Table 7.8: Number of calls to the *propagate* function for each threshold, for the base and small models

In relation to solution density, as shown in Table 5.3, we can also observe that these six instances correspond to those with a higher solution density, supporting the idea that they are therefore easier instances to solve.

7.2.3 Random Model Sampled from the Baseline Distribution

The graphs showing, for the training instances, the number of solutions, nodes, and failures computed with the random model following the base model’s distribution, for different thresholds are found in Figure 11.26, Figure 11.27, and Figure 11.28. The same graphs obtained on the test instances are found in Figure 11.29, Figure 11.30, and Figure 11.31.

As expected, this model performs much worse than both the base model and the small model. Indeed, for the training instances, the model does not find any solution for many of them, regardless of the threshold. The only training instances for which the model is able to find solutions are the six instances presented earlier that have the highest solution density. However, the number of solutions found for each threshold is lower than the number of solutions found by the baseline model and the small model. For the test instances, this random model only manages to find solutions for one instance among those with multiple solutions. This is the *MagicSquare-03-sum* instance, which has the highest solution density among the test instances, as shown in Table 5.3.

As illustrated in Figure 7.14, for instances with a single solution, the model succeeds in finding the solution randomly for only a small number of them. Therefore, even if the random model happens to find the unique solution for a given threshold, there is no guarantee that it will find it again for a larger threshold.

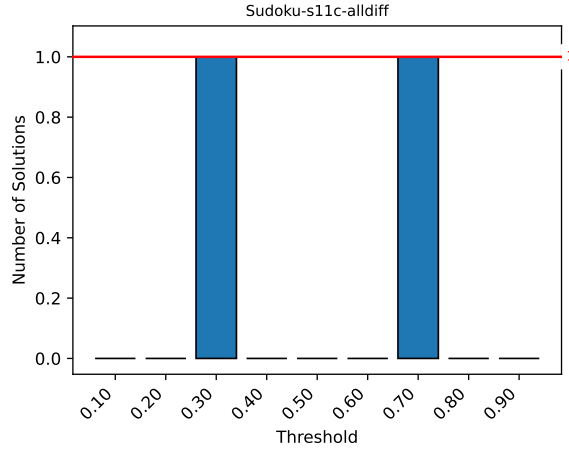


Figure 7.14: Number of solutions as a function of the threshold for the *Sudoku-s11c-alldiff* instance when using the random model following the base model’s distribution

When looking at the number of nodes or failures for a fixed instance, we observe a completely random number of nodes and failures. However, we can notice for some instances a slight upward trend within this random pattern. This is expected behavior, since the higher the threshold increases, the more likely the model is to draw a number below that threshold, and therefore to keep the edge.

7.2.4 Fully Random Model

As explained during the previous phase of metric collection, since this model does not rely on any training data, the training and test instances were combined into a single large batch. Appendices Figure 11.32, Figure 11.33, and Figure 11.34 respectively show the number of solutions, nodes, and failures when using the random model with a uniform distribution for both the training and test instances.

As expected, this model is the worst of all, finding no solutions for 35 of the 37 instances. The first exception corresponds to the *Allergy* instance, where the model succeeds, like the other models, in finding the unique solution regardless of the threshold. This can be explained by the structure of the problem, which requires a very small number of nodes and calls to the propagate function when using the classical propagator without machine learning. Indeed, for this instance, the classical solver without machine learning builds only two nodes and calls the propagate function five times. With the random model, the propagate function is called only three times with two nodes built. Across all test and training instances, when using the classical solver without machine learning, this is the instance that calls the propagate function the fewest times.

The second instance where this solver succeeded in finding a solution is the *Purdey* instance. This instance has a unique solution, and the solver managed to find it randomly only for the threshold of 0.4. It is the second instance with the fewest calls to the propagate function when using the classical solver without machine learning, calling it ten times and building two nodes. The fact that the solver with the random propagator finds the unique

solution for the threshold of 0.4 is clearly a matter of luck, given the low number of calls to the propagate function.

It is interesting to note that, in addition to calling the *propagate* function only a few times, as illustrated in [Table 5.3](#), the *Allergy* and *Purdey* instances are those with the highest solution density among the training and test instances, reinforcing the idea that they are the easiest problems to solve.

The third instance with the lowest number of calls to the propagate function is the *Kakuro-hard-010-sumdiff* instance, which triggers 375 calls to the propagate function when using the classical solver. The uniform-random solver did not benefit from sufficient luck to avoid a fatal misclassification across all propagate calls and, consequently, was unable to find a solution.

Chapter 8

Runtime Analysis

Although the evaluation of efficiency and execution time falls outside the scope of this thesis, it remains relevant to understand the challenges and positioning of the proposed method. This section therefore aims to compare the execution time of the different approaches with that of the classical solver that does not rely on machine learning.

8.1 Hardware Configuration

The execution time experiments performed on the classical solver as well as on the solvers using machine learning models were run on the high-performance computing centers of the **CECI** (Consortium des Équipements de Calcul Intensif) [32]. More precisely, the *Lyra* cluster was used. This is a cluster hosted at ULB (Université Libre de Bruxelles), specifically designed for intensive machine learning applications. More precisely, the cluster is composed of 40 compute nodes. Each virtual node has access to one Nvidia RTX 6000 ADA GPU with 48 GB of GDDR6 memory (18,176 FP32 CUDA cores), a 32-core CPU based on the AMD EPYC-Genoa microarchitecture, and 128 GB of RAM. During the experiments, 4 CPUs were used with 16 GB of RAM.

8.2 Runtime Measurement

As a reminder, the solver runs in a Java environment, while the propagator using the machine learning model runs in Python. Communication between the solver and the propagator is carried out through a socket. Moreover, as explained in [subsection 5.4.2](#), the checker is based on a *SateInt* and an index array in order to reduce the propagator’s execution time within Java.

The text below compares the runtime of the solver without machine learning to the runtime of the different solvers equipped with machine-learning-based propagators. For each model, the appendices provide a table containing, for each training and test instance, the total execution time of the solver without machine learning, the total execution time of the solver using the machine learning model, the time spent in Java when using the machine learning model, and the time spent in Python when using the machine learning model. For each instance, the reported runtime measurements for the solver using the machine learning model

correspond to a configuration in which the model threshold is fixed to 0.5. In addition, another column provides information regarding the ratio between the total runtime of the solver with machine learning and the total runtime of the solver without machine learning. A final column reports the ratio between the time spent in the machine-learning-based propagator (Python code) and the number of calls to the propagate function. This last column makes it possible to obtain an average execution time per call to the propagate function.

As explained above, Appendices [Table 11.10](#) and [Table 11.11](#) present the execution times when using the base model on the training and test instances, respectively. Appendices [Table 11.12](#) and [Table 11.13](#) report the same measurements when using the small model. Appendices [Table 11.14](#) and [Table 11.15](#) highlight the random model following the base model’s distribution. Finally, Appendix [Table 11.16](#) corresponds to the uniform-random model on the combined training and test instances.

[Table 8.1](#) reports, for each model, the sum of the execution times over all training and test instances for each timing measurement when the threshold is fixed to 0.5.

Model	Solver without AI total (ms)	AI total (ms)	AI Java (ms)	AI Python (ms)
A	185122	15983630.00	166735.00	14272715.00
B	185122	30539963.00	217470.00	27202558.00
C	185122	44626.00	4190.00	19101.00
D	185122	494.00	266.00	90.00

Table 8.1: Global execution time comparison summed over all training and test instances, between the solver without AI and the AI-based solvers with a threshold of 0.5 (**A**: baseline model, **B**: small model, **C**: random model following the baseline model’s distribution, **D**: uniform random model)

[Table 8.2](#) summarizes, for each model, the ratio between the total execution time of the solver using the machine learning model and the total execution time of the solver without machine learning, when summing the runtime of all training and test instances. In addition, a final column reports the ratio between the total time spent in the machine learning model (Python code) across all instances and the total number of calls to the propagate function. This column provides an average execution time for a single call to the propagate function.

Model	AI total / Solver without AI total	Python AI / # propagation (ms)
A	86.34	0.496204
B	164.97	0.452957
C	0.24	0.028168
D	0.00	0.517241

Table 8.2: Speedup and average propagation time in Python for the AI-based solver, summed over all training and test instances with a threshold of 0.5 (**A**: baseline model, **B**: small model, **C**: random model following the same distribution as the baseline model, **D**: uniform random model)

First, referring to the appendices and to [Table 8.1](#), we observe that the total execution time when using the AI model is significantly higher than the execution time of the solver without machine learning. Conversely, for both random models, the total time is lower than that of the solver without machine learning. We also note that, for the two AI models, the solver spends most of its time in the Python code and therefore in edge prediction.

This first observation does not allow us to draw conclusions regarding the efficiency of the propagator using the AI model. Indeed, the number of explored nodes, and consequently the number of calls to the propagate function, is not the same in the solver with and without machine learning. If the solver using the machine learning model builds more nodes than the base solver, it is more likely that the runtime will be higher for the solver with machine learning.

It is therefore relevant, by referring to the appendices and to [Table 8.2](#), to examine the column showing the ratio between the time spent in the AI code and the number of calls to the propagate function. We observe that the two AI models and the uniform random model exhibit, on average over all instances, a relatively similar propagation time (0.496ms for the base model, 0.453ms for the small model, and 0.517ms for the uniform random model). The model using the random distribution based on the base model is, on average, the fastest of all, with a propagation time of 0.028 ms. This observation shows that although the completely random model is, in terms of total solving time, the fastest to complete the instances, its propagator execution time remains close to that of the two AI models. Moreover, it is interesting to note that there is no significant difference in propagator execution time between the base model and the small model. Indeed, since the code relies on vectorized instructions, the size of the vector has only a limited impact on inference time. It is important to note that even though the total time required to solve all instances is lower with the baseline model than with the small model, this does not allow us to draw any conclusions, since the small model explores, on average, more nodes than the baseline model.

Finally, as explained later in [section 10.1](#), the first observation in [Table 8.1](#) highlights an initial limitation of the method with the *AllDifferent* constraint. Indeed, the classical solver without machine learning manages to find all solutions to the problems in a shorter time than the solver using machine learning.

Chapter 9

Constraint Optimization Problems

Another objective of this thesis is to investigate constraint optimization problems. The goal is to compare the best solution found by the classical solver with the best solution obtained when using machine-learning-based propagators.

9.1 Theory

A constraint optimization problem (COP) is defined in the same way as a classical CSP, with the addition of a function f that links the variables and assigns a score to a solution. Thus, a COP can be defined as a quadruple $P = (X, D, C, f)$ with

- $X = \{X_1, X_2, \dots, X_n\}$ a set of variables.
- $D = \{D_1, D_2, \dots, D_n\}$ the corresponding domains, with D_i being the domain of variable X_i .
- $C = \{C_1, C_2, \dots, C_m\}$ a set of constraints.
- $f : D_1 \times D_2 \times \dots \times D_n \rightarrow \mathbb{R}$ an objective function to be optimised (maximised or minimised).

A COP is solved in the same manner as a CSP with respect to search tree construction and constraint propagation. The main difference lies in the addition of a new constraint each time a new solution is found. Indeed, when the i -th solution σ_i is identified, the associated objective function value $f(\sigma_i)$ is evaluated. A new constraint is then posted to ensure that any subsequent solution must improve upon the current one. More precisely, when solution σ_i is found, we add the constraint

$$C_i : f(\sigma_j) < f(\sigma_i) \quad \forall \quad j > i \quad (\text{if minimizing}) \quad (9.1)$$

or

$$C_i : f(\sigma_j) > f(\sigma_i) \quad \forall \quad j > i \quad (\text{if maximizing}) \quad (9.2)$$

Consequently, the most optimal solution according to the objective function is the last solution found.

Example 9.1. Let us assume two variables X and Y with domains $X = \{1, 2, 3, 4\}$ and $Y = \{1, 2, 3, 4\}$, with no constraints, and an objective function $f = X + Y$ to be minimized.

If the solver finds a first solution with $X = 1$ and $Y = 2$, it evaluates the objective function as $f = 1 + 2 = 3$ and adds a "less than" constraint enforcing the inequality $X + Y < 3$.

This new constraint reduces the search tree by activating the propagator of the "less than" constraint. Thus, although there are 16 distinct solutions to this problem, the solver will find only one additional solution that is better than the first ($X = 1$ and $Y = 1$).

9.2 Problems Under Study

In order to compare the quality of the solutions found by the classical solver with those obtained using the machine-learning-based propagator, different COP instances were selected from the [XCSP3](#) website [27]. All of the selected instances are problems whose objective is the minimization of the objective function. Clearly, this does not reduce generality due to the symmetry of the constraint that will be imposed to improve the solutions each time a new one is found. As with the CSP problems, the branching technique used is *First fail* [28]. [Table 9.1](#) provides information related to these instances when the optimization function is removed (transforming the COP into a CSP) and they are solved with the classical solver. The *# AllDiff prop.* column provides information on the number of calls to the *propagate* function in the *AllDifferent* constraint. In addition, the density column reports the solution density as defined in [Equation 5.1](#), computed after removing the optimization objective.

Instance Name	# Solutions	# Nodes	# Failures	# AllDiff prop.	Density
GolombRuler	3786	7946	188	14184	3.04×10^{-10}
RubiksCube	—	—	—	—	—
PathFinding	2457	4912	0	7278	6.96×10^{-13}
ClockTriplet	112327660	224664246	4464	103246522	3.71×10^{-7}
Pyramid	8076	17864	857	40607	4.59×10^{-7}
Charlotte	28608	57220	3	131066	1.15×10^{-6}
KMedian	1	2	1	8	1.73×10^{-134}
Pyramid-2	334	3346	1340	10518	4.08×10^{-13}

Table 9.1: Statistics for the selected instances without the objective function

As shown in [Table 9.1](#), even after running the *RubiksCube* instance for three days with the classical solver without an objective function, the solver was unable to enumerate all possible solutions. This reinforces the argument in favor of the efficiency gained by adding a constraint after each new solution when the problem is transformed into a COP.

[Table 9.2](#) summarizes the information related to the instances when the optimization function is included (kept as a COP) and solved with the classical solver. The *Objective Value* column indicates the value of the objective function for the best (last) solution found, and the *# AllDiff prop.* column provides information on the number of calls to the *propagate* function in the *AllDifferent* constraint.

Instance Name	# Solutions	Objective Value	# Nodes	# Failures	# AllDiff prop.
GolombRuler	2	6	6	2	11
RubiksCube	14	1	378	176	222
PathFinding	3	8	280	138	749
ClockTriplet	13	21	87232	43604	183224
Pyramid	1	8	46	23	99
Charlotte	21	757	1204	582	3509
KMedian	1	1472	2	1	8
Pyramid-2	1	20	592	296	1616

Table 9.2: Statistics for the selected instances with the objective function

It is interesting to note the large difference in the number of solutions found when the optimization function is removed (CSP) compared to when it is included (COP). Moreover, the significant difference in the number of nodes between the CSP and COP formats also suggests the strong efficiency of adding a new constraint after each newly discovered solution.

The *Number of best solutions* column in Table 9.3 reports the number of solutions that are evaluated as optimal by the objective function. In other words, these are the solutions for which the objective function assigns the same value, and this value corresponds to the best possible one. The *Best-solution density* column represents the ratio between the number of solutions sharing the same optimal value and the total number of solutions when the objective function is removed. This measure makes it possible to assess whether it is difficult to find a solution of the same quality as the best one, or whether a large number of solutions achieve the same optimal quality.

Instance Name	Number of best solutions	Best-solution density
GolombRuler	2	5.28×10^{-4}
RubiksCube	-	-
PathFinding	9	3.66×10^{-3}
ClockTriplet	261	2.32×10^{-6}
Pyramid	2	2.48×10^{-4}
Charlotte	2	6.99×10^{-5}
KMedian	1	1
Pyramid-2	6	1.80×10^{-2}

Table 9.3: Number of best solutions and best-solution density for the selected instances

We can observe a wide diversity in the *Best-solution density* column, with a particular case for *KMedian*. Indeed, when this instance is run with the objective function removed, the solver finds only a single solution. This solution is therefore trivially the best one, resulting in a unitary ratio.

Appendix Figure 11.35 shows the evolution of the objective function as different solutions are progressively found. Figure 9.1 presents the same plots for the three instances that yielded the highest number of solutions during the search.

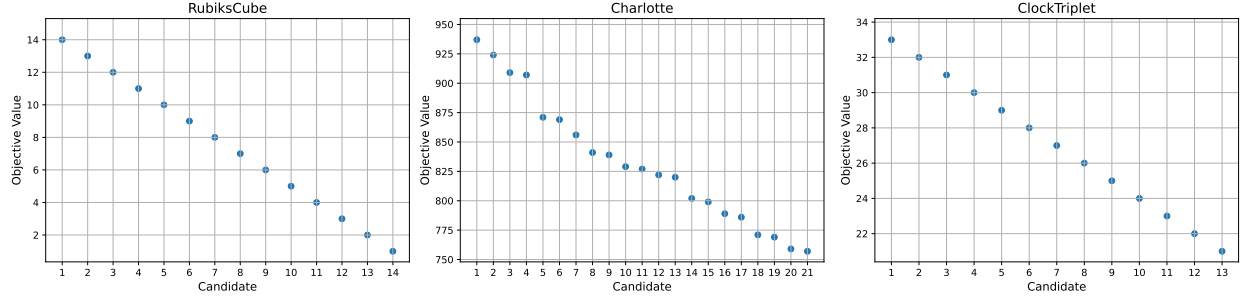


Figure 9.1: Evolution of the objective function value over the different solutions found for the *RubiksCube*, *Charlotte*, and *ClockTriplet* instances

As expected, the solver finds solutions whose objective function value decreases progressively as new solutions are discovered.

Figure 9.2 shows the evolution of the objective value of the best solution found over time for the three instances illustrated in Figure 9.1.

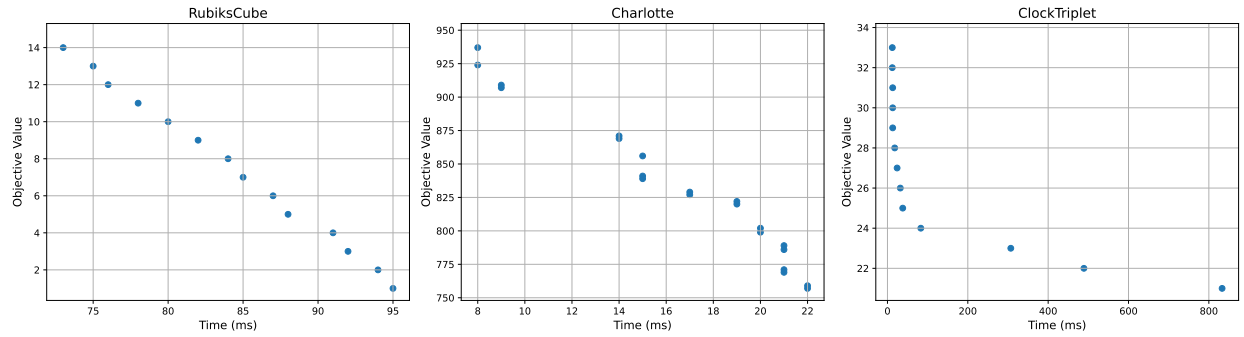


Figure 9.2: Evolution of the objective function value of the best solution found over time for the *RubiksCube*, *Charlotte*, and *ClockTriplet* instances

It is interesting to observe that although the best solution value evolves linearly for the *RubiksCube* and *ClockTriplet* instances, the shape of the evolution differs significantly when analyzed with respect to execution time. The *RubiksCube* instance continues to follow a linear trend, whereas for *ClockTriplet*, the best solution value decreases very rapidly before stabilizing. In the case of *ClockTriplet*, one may assume that the addition of a new constraint after each solution strongly restricts the problem, making the discovery of subsequent solutions increasingly difficult.

The instances were also executed by transforming the minimization problems into maximization problems. This transformation makes it possible to obtain the worst solution to the problem and, through the objective function, to derive a lower bound for the best solution and an upper bound for the worst solution. Table 9.4 summarizes, for each instance, the objective function evaluation of the best and worst solutions found using the classical solver without machine learning.

Instance Name	Min val. (Best sol.)	Max val. (Worst sol.)
GolombRuler	6	20
RubiksCube	1	20
MultiAgentPathFinding	8	12
ClockTriplet	21	33
Pyramid	8	50
Charlotte	757	1634
KMedian	1472	1472
Pyramid-2	20	30

Table 9.4: Objective function value for the best and worst solution of each instance

9.3 Experiments

In this section, the different instances described above are executed using the solver with the *AllDifferent* propagator based on the machine learning model. In the graphs showing the evolution of the objective value of the best solution as a function of the threshold, the red line corresponds to the objective function evaluation of the best solution found when using the classical propagator. The dashed blue line corresponds to the objective function evaluation of the worst solution when using the classical propagator.

9.3.1 Baseline Model

Appendices [Figure 11.36](#), [Figure 11.37](#), [Figure 11.38](#), [Figure 11.39](#), and [Figure 11.40](#) respectively show the evolution, as a function of the threshold, of the number of solutions found, the number of failures, the number of nodes, the objective function value of the best solution found, and the number of calls to the *AllDifferent* propagator when using the baseline model. Below are some graphs that are particularly interesting to interpret.

By comparing the number of nodes and the objective function value across different thresholds, we observe a tangible benefit of our approach. As illustrated in [Figure 9.3](#), the left figure shows the evolution of the number of nodes as a function of the threshold when using the machine-learning-based propagator. In the same figure, the red line represents the number of nodes when running the instance with the classical solver. The right figure shows the evolution of the objective value of the best solution found for each threshold. We can observe that, despite using a small threshold and thus heavily pruning the search tree, the model is still able to find a solution that achieves the same objective value as the classical solver. For instance, by setting the threshold to 0.3, we obtain 64 nodes, less than one third of the number explored by the classical solver, while still producing a solution of equivalent quality.

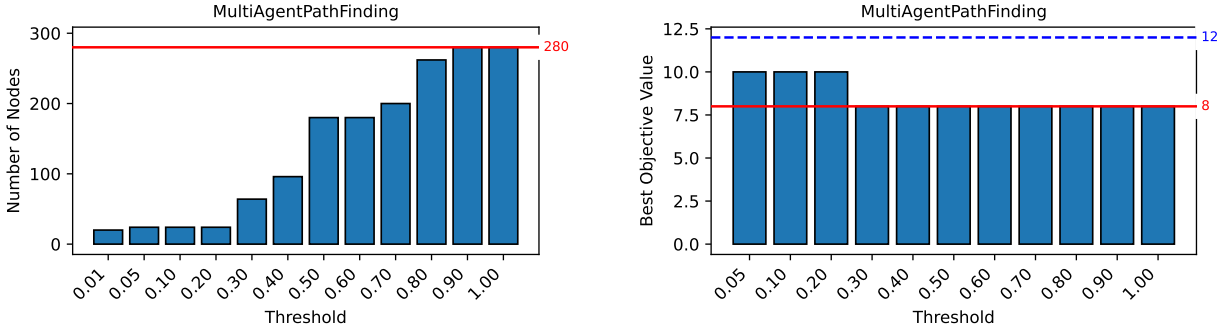


Figure 9.3: Comparison of the number of nodes (left) and the evolution of the objective function value evaluated on the best solution (right) as a function of the threshold for the *MultiAgentPathFinding* instance

It is important to note in Figure 9.3 that the right-hand figure does not contain data for the threshold value of 0.01. Indeed, as shown in Figure 9.4, the solver powered by our model was unable to find a solution for this threshold, which is too aggressive. Consequently, it is not possible to evaluate the objective function on a nonexistent solution.

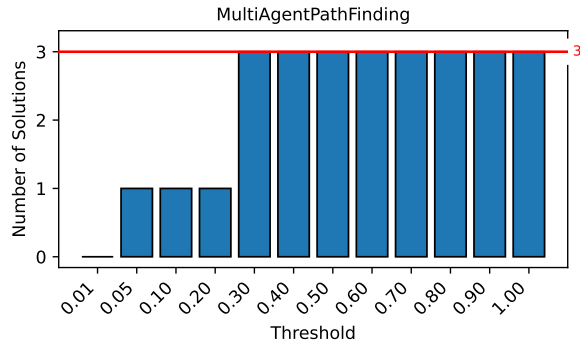


Figure 9.4: Number of solutions found versus threshold using the baseline model for the *MultiAgentPathFinding* instance

Figure 9.5 presents another instance for which similarly satisfactory results are achieved.

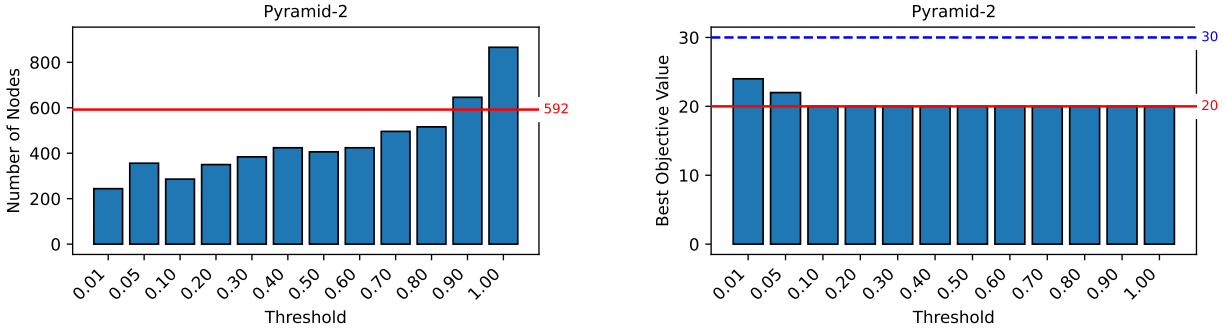


Figure 9.5: Comparison of the number of nodes (left) and the evolution of the objective function value evaluated on the best solution (right) as a function of the threshold for the *Pyramid-2* instance

It is also worth noting that this satisfactory behavior is not observed for all instances. As shown in Figure 9.6, in this case, although a solution of equivalent quality to that found by the classical solver is obtained for each threshold, the number of explored nodes is, for every threshold, higher than the number explored by the classical solver.

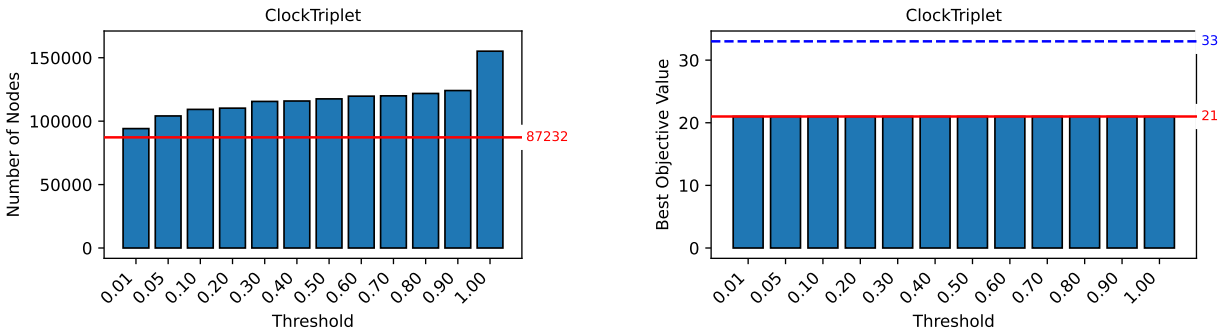


Figure 9.6: Comparison of the number of nodes (left) and the evolution of the objective function value of the best solution (right) as a function of the threshold for the *ClockTriplet* instance

This model is therefore, in certain cases, useful for obtaining a solution of comparable quality to that of a classical solver while often drastically reducing the number of explored nodes.

9.3.2 Small Model

Appendices Figure 11.41, Figure 11.42, Figure 11.43, Figure 11.44, and Figure 11.45 respectively show the evolution, as a function of the threshold, of the number of solutions found, the number of failures, the number of explored nodes, the objective function value of the best solution found, and the number of calls to the *AllDifferent* propagator when using the small model. Below are several graphs that are particularly interesting to interpret.

First of all, as illustrated in Figure 9.7, we can see, except for one exception, that for a fixed threshold the baseline model is generally better at finding solutions than the small model. However, when using thresholds greater than 0.2, both models are able to find the same number of solutions as the solver without machine learning.

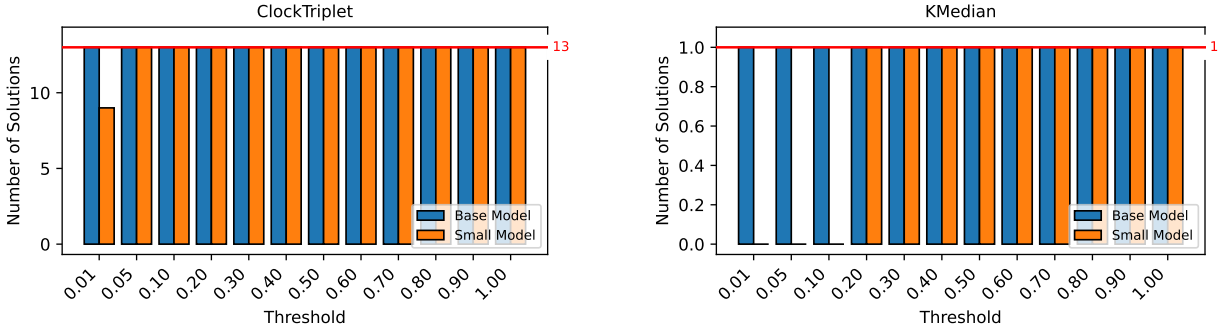


Figure 9.7: Comparison of the number of solutions obtained using the baseline model and the small model for the *ClockTriplet* instance (left) and the *KMedian* instance (right) as a function of the threshold

Regarding the number of explored nodes, it is difficult to draw a strict conclusion about whether the small model explores more or fewer nodes than the baseline model. The number of explored nodes is often similar, with three exceptions for the instances *MultiAgentPathFinding*, *ClockTriplet* and *Pyramid*, where the number of explored nodes is significantly higher when using the small model.

In addition to finding solutions for small thresholds when using the baseline model, as illustrated in Figure 9.8, for a fixed small threshold, and with one exception, the solutions found by the baseline model are of higher quality compared to the quality of the solutions found by the small model. However, it is important to qualify this statement by noting that for the majority of the thresholds studied, the quality of the solutions found by the baseline model and the small model is equivalent.

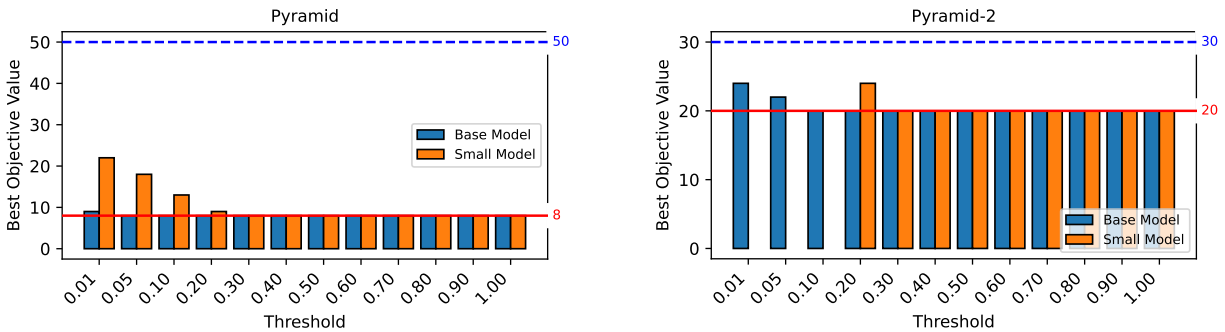


Figure 9.8: Comparison of the quality of the solutions found between the baseline model and the small model, for the *Pyramid* instance (left) and *Pyramid-2* (right)

The most notable difference observed for these COP instances between the baseline model

and the small model lies in the baseline model’s ability to find solutions for small thresholds, a capability that the small model does not exhibit.

9.3.3 Random Model Sampled from the Baseline Distribution

Appendices [Figure 11.46](#), [Figure 11.47](#), [Figure 11.48](#), [Figure 11.49](#) and [Figure 11.50](#) respectively show the evolution, as a function of the threshold, of the number of solutions found, the number of failures, the number of nodes, the value of the objective function of the best solution found, and the number of calls to the *AllDifferent* propagator when using the random model following the distribution of the baseline model. Below are some graphs that are interesting to interpret.

First of all, as for standard CSP instances, the number of solutions found when the threshold increases does not follow an increasing trend. Moreover, as illustrated in [Figure 9.9](#), it may be surprising that for some thresholds and some instances, the number of solutions found is higher than the number of solutions found when using the solver without machine learning. However, this behavior is not impossible, since when the solver solves a COP problem, it does not enumerate all solutions. Indeed, as explained previously, when a solution is found, the solver adds a constraint requiring the next solution to be of better quality.

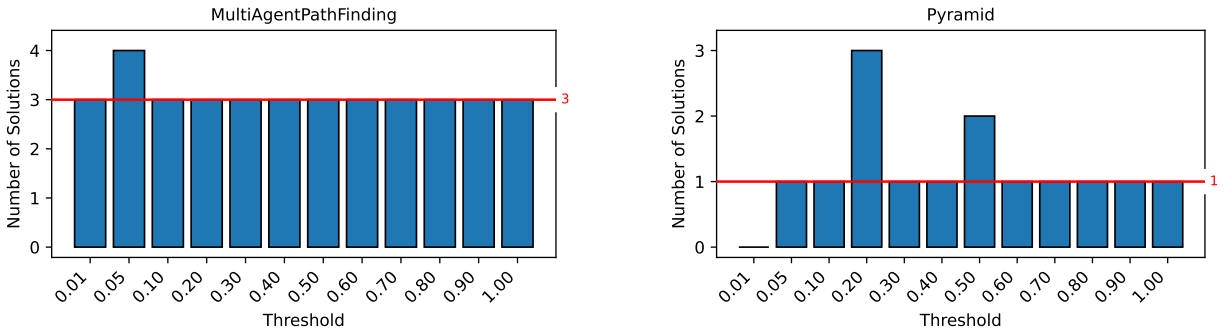


Figure 9.9: Comparison of the number of solutions found between the random model following the distribution of the baseline model and the solver without machine learning (red line), for several thresholds, for the instances *MultiAgentPathFinding* (left) and *Pyramid* (right)

When considering only the number of explored nodes for the different instances in comparison with the number of nodes explored when using the solver without machine learning, no clear conclusion can be drawn. Indeed, for some instances, the number of explored nodes exceeds that of the solver without machine learning, while for other instances it is lower. However, for some instances, when comparing the number of explored nodes with the value of the objective function of the best solution found, certain instances are interesting to analyze.

As shown in [Figure 9.10](#), for the *ClockTriplet* instance, for some thresholds, the solver explores fewer nodes than the solver without machine learning, while finding solutions of similar or even equivalent quality to the quality of the best solution found by the solver without machine learning.

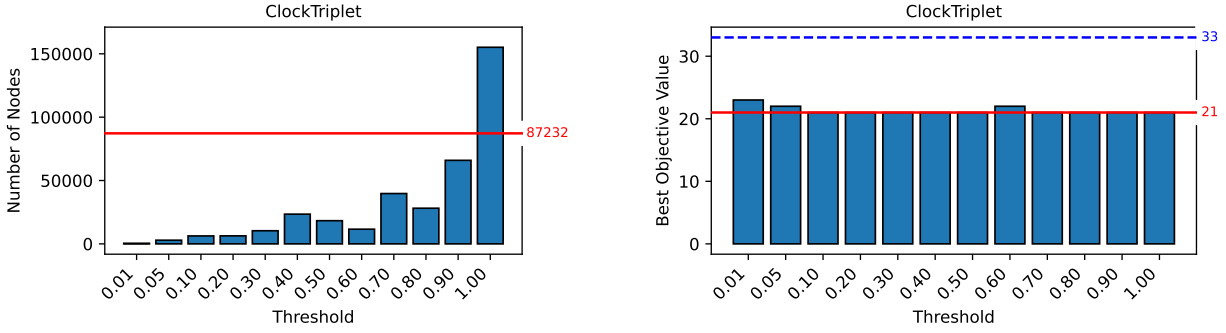


Figure 9.10: Comparison of the number of nodes (left) and the value of the best solution found (right) for each threshold for the *ClockTriplet* instance using the random solver following the distribution of the baseline model

As shown in Figure 9.11, the expected behavior illustrated above is not always observed. Indeed, for the *Pyramid-2* instance, although the number of explored nodes remains lower than the number of nodes explored by the solver without machine learning, the solver does not manage to find solutions of the same quality as those found with the solver without machine learning.

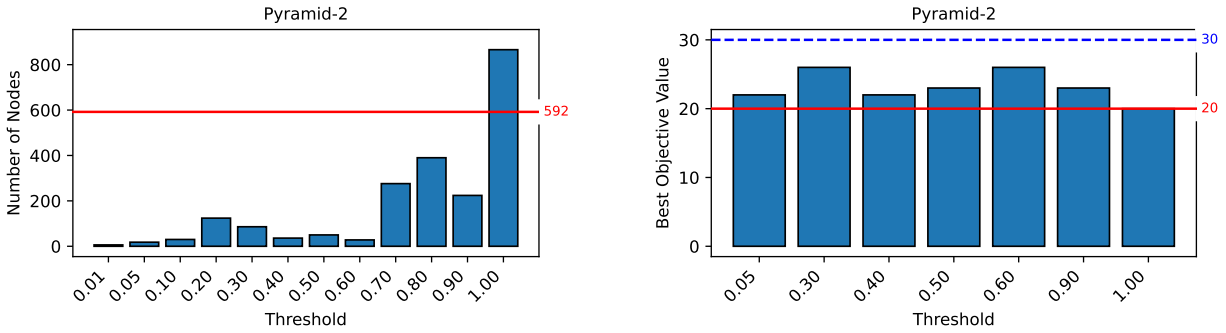


Figure 9.11: Comparison of the number of nodes (left) and the value of the best solution found (right) for each threshold for the *Pyramid-2* instance using the random solver following the distribution of the baseline model

More disappointingly, as shown in Figure 9.12 for the *GolombRuler* instance, although the solver finds solutions of the same quality as those found by the solver without machine learning for all thresholds, the number of explored nodes is higher than the number of nodes explored by the solver without machine learning.

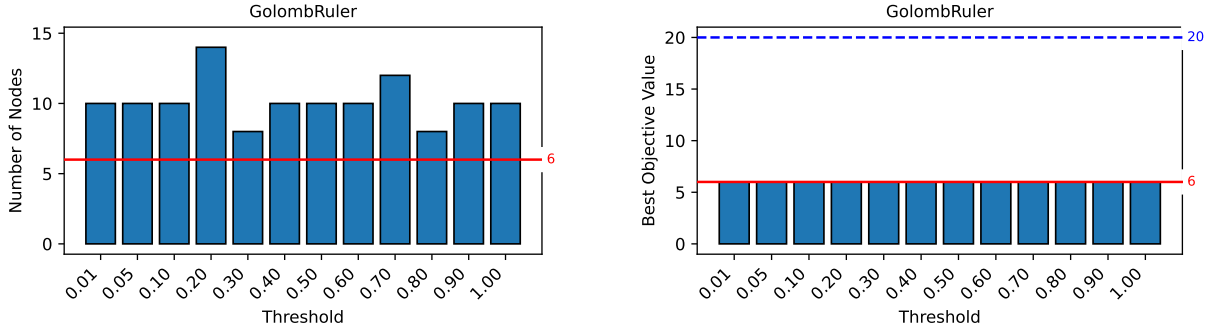


Figure 9.12: Comparison of the number of nodes (left) and the value of the best solution found (right) for each threshold for the *GolombRuler* instance using the random solver following the distribution of the baseline model

9.3.4 Fully Random Model

Appendices [Figure 11.51](#), [Figure 11.52](#), [Figure 11.53](#), [Figure 11.54](#), and [Figure 11.55](#) respectively show the evolution, as a function of the threshold, of the number of solutions found, the number of failures, the number of nodes, the objective function value of the best solution found, and the number of calls to the *AllDifferent* propagator when using the random model with a uniform distribution. Below are several graphs that are particularly interesting to interpret.

First, as illustrated in [Figure 9.13](#), it is interesting to note that for the majority of the instances (6 out of 8), the model explores a very small number of nodes for thresholds below 0.8. We can also observe, again for the same majority of instances, that the number of explored nodes increases drastically when the threshold is set to 1. Both behaviours are expected: when the threshold moves away from 1, the model starts to reduce the domains randomly, thus cutting branches at random, which quickly leads to inconsistencies and therefore fewer explored nodes. Moreover, when the threshold is set to 1, all randomly drawn numbers will be less than or equal to the threshold, and therefore all domain values will be preserved. This corresponds to a special case in which the COP is solved without any propagation in the *AllDifferent* constraint.

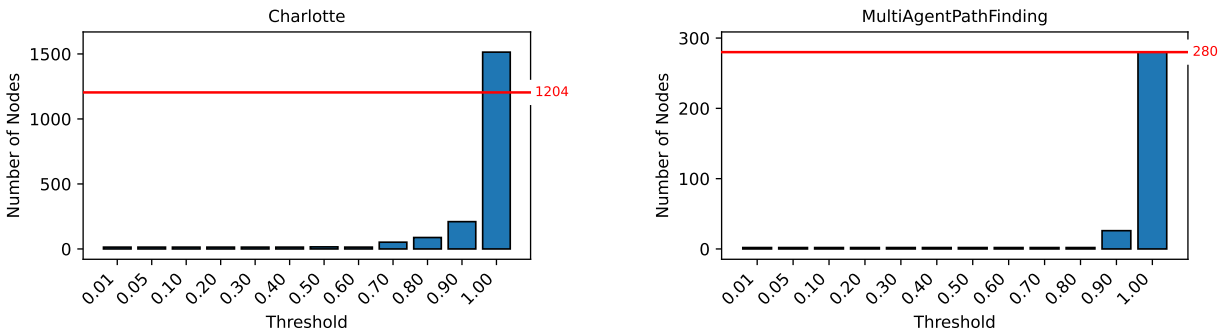


Figure 9.13: Illustration of the number of explored nodes for the *Charlotte* instance (left) and the *MultiAgentPathFinding* instance (right) as a function of the threshold

As illustrated in Figure 9.14, the number of solutions found is mostly zero for thresholds less than or equal to 0.6 (7 instances out of 8). Moreover, the number of solutions found when the threshold is set to 1 is always the same as the number of solutions found with the baseline model. Both phenomena can be explained by the same reasoning presented earlier. It is worth noting that for 3 out of the 8 instances, the random model was unable to find a single solution unless the threshold was set to 1.

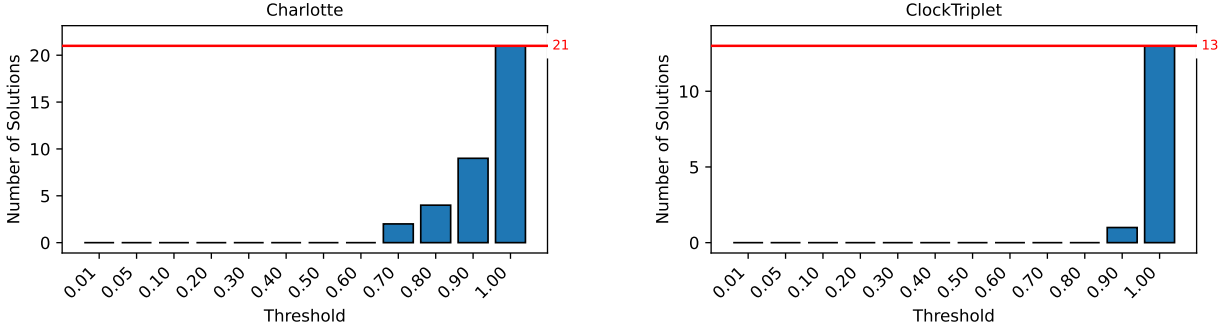


Figure 9.14: Illustrations of the number of solutions found for the *Charlotte* instance (left) and the *ClockTriplet* instance (right) as a function of the threshold

Following the previous explanations, we can also observe in Figure 9.15 that the bar plot of the number of calls to the *propagate* function of the *AllDifferent* constraint follows the same pattern as the number of nodes with respect to the threshold. Indeed, if the number of explored nodes decreases, the model has fewer opportunities to call the *propagate* function within those nodes.

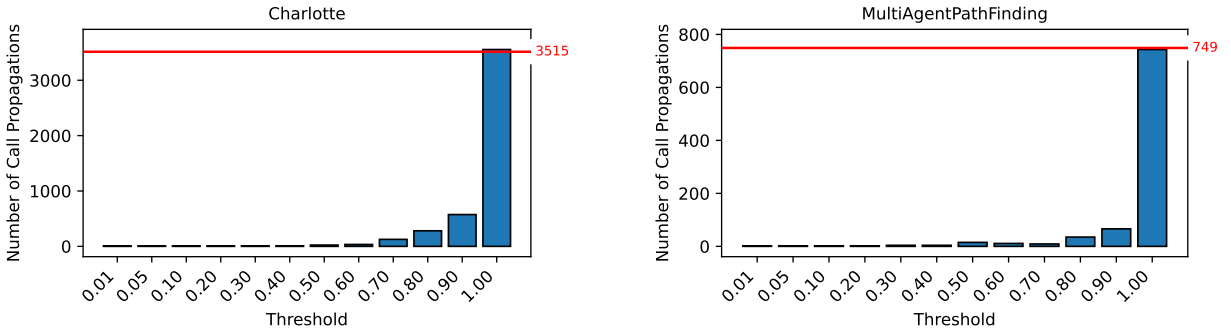


Figure 9.15: Illustrations of the number of calls to the *propagate* function in the *AllDifferent* constraint for the *Charlotte* instance (left) and the *MultiAgentPathFinding* instance (right) as a function of the threshold

An interesting comparison concerns the evolution of the objective function value for different thresholds with respect to the number of explored nodes. Figure 9.16, Figure 9.17, and Figure 9.18 highlight this comparison for three instances. Only the thresholds that produced at least one solution are shown in the bar plot of the objective function.

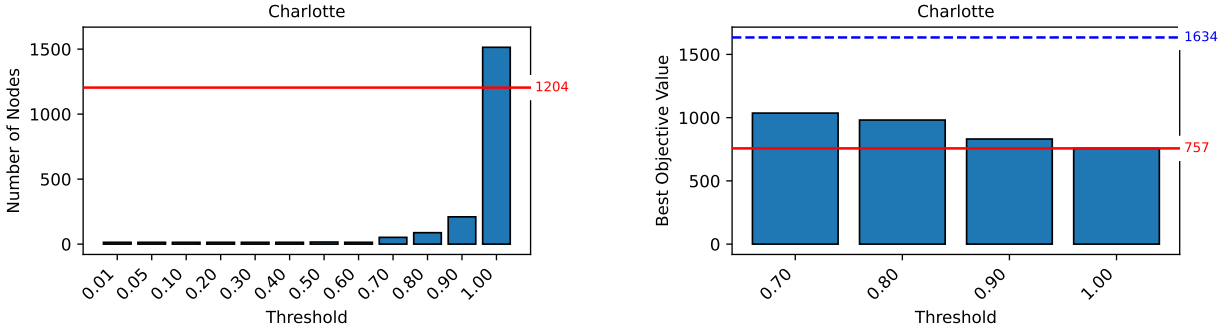


Figure 9.16: Comparison of the number of nodes (left) with the objective function value of the best solution (right) as a function of the threshold for the *Charlotte* instance

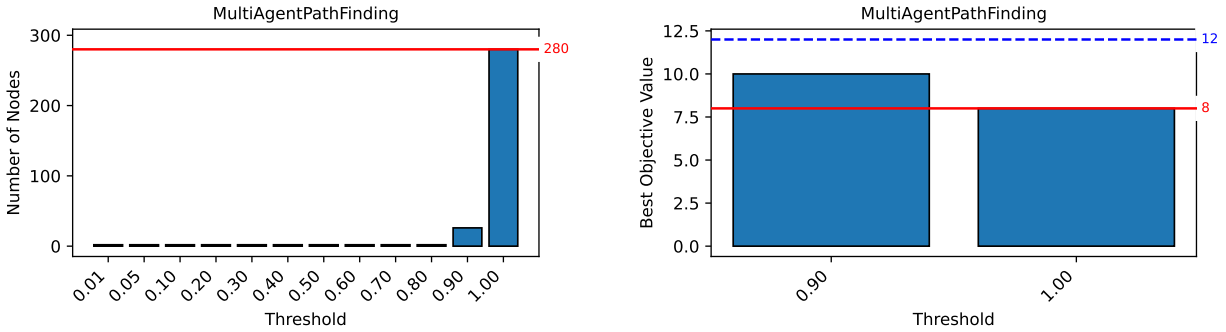


Figure 9.17: Comparison of the number of nodes (left) with the objective function value of the best solution (right) as a function of the threshold for the *MultiAgentPathFinding* instance

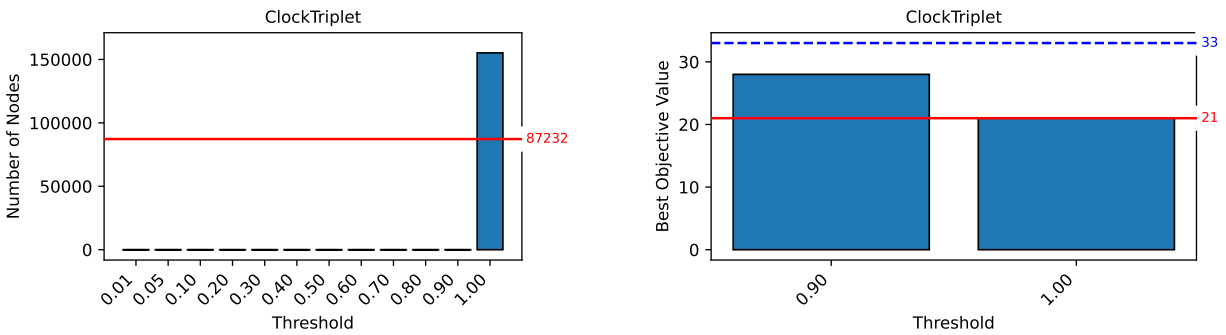


Figure 9.18: Comparison of the number of nodes (left) with the objective function value (right) as a function of the threshold for the best solution of the *ClockTriplet* instance

For these instances, with the exception of the threshold value of 1, we observe that even with a very small number of explored nodes, the random model is still capable of producing solutions whose quality is close to that of the optimal one. Thus, this model can, in certain

cases, be of interest when the goal is to obtain a solution that is slightly suboptimal yet reasonably close to what a classical solver would find, while exploring only a very limited portion of the search space.

Chapter 10

Limitations and Perspectives

10.1 Limitations

As explained in the execution time analysis, the higher runtime observed when using the machine-learning-enhanced solver compared to the solver without machine learning constitutes a first limitation of the approach. However, this thesis provides a proof of concept of the method on the *AllDifferent* constraint. It is evident that the *AllDifferent* propagator is not the one with the highest temporal complexity. The method would therefore become more relevant when applied to propagators of other constraints with higher algorithmic complexity, such as scheduling constraints or cumulative constraints.

Moreover, although the solver spends most of its time inside the Python code, the use of a Java solver communicating with the propagator in Python through a socket also slows down the propagation process. In this regard, it would be interesting to integrate both the solver and the propagator into a single framework following industrial standards.

Finally, since the constraint can be represented as a graph, it appeared natural to use a machine learning model based on GNNs. Although the operations are vectorized to increase propagation speed, it may be worthwhile to consider other, more efficient architectures and to find a better balance between efficiency and model quality¹.

10.2 Future Research Perspectives

It is evident that the approach proposed for the *AllDifferent* constraint can be applied to the propagators of other constraints commonly used in constraint programming. It would then be interesting, in addition to measuring the impact of each constraint independently, to study how different constraints using the model can together reduce the search tree even more effectively.

Moreover, the machine learning model is invoked at every call to the propagator. As observed in the training data analysis, in the vast majority of cases no domain is reduced during propagation. In this sense, it seems relevant to introduce a small and very fast *load balancer* model that estimates whether it is worthwhile to call the larger prediction model,

¹Other architectures based on search trees, tables, etc., were tested and can be found in the *other_try* folder of the GitHub repository

another smaller model, the simple checker, or to do nothing at all. Depending on the efficiency of such a *load balancer*, one could expect a reduction in the time spent inside the propagator by invoking the large, slower model only when necessary.

Chapter 11

Conclusion

In conclusion, this thesis has introduced an original proof of concept for a propagator leveraging a machine learning algorithm. The work began with an exploration of the functioning of a constraint programming solver, followed by an analysis of the *AllDifferent* constraint, with a focus on the two main propagators associated with it. We then defined and examined the various machine learning tools required to construct the proposed method, and reviewed the state of the art related to the envisioned approach. Subsequently, we designed several architectures for the machine-learning-based propagator and conducted experiments on a set of problems modeled as CSPs.

The various experiments demonstrated the usefulness of the method through the reduction of the explored search tree while still preserving a large number of solutions. The method also proved to be effective when applied to constraint optimization problems, successfully maintaining solutions of the same quality as those found by the classical solver, while exploring only a subset of the nodes.

We also highlighted the limitations of the proposed method, in particular the time spent within the Python based model to obtain predictions for the graph edges, which remains a significant challenge. Nevertheless, the discussion on future perspectives has motivated the application of the method to more complex propagators, as well as the exploration of additional strategies, most notably the introduction of a *load balancer* capable of deciding which model to invoke, or whether to invoke any model at all, during propagation.

References

- [1] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. 3rd ed. Hoboken, NJ: Pearson, 2010. ISBN: 978-0136042594.
- [2] P. Schaus. *LINFO2266: Advanced Algorithms for Optimization*. https://linfo2266.readthedocs.io/en/latest/_static/slides/07-intro-cp.pdf. UCLouvain. 2025.
- [3] W.-J. van Hoeve. “The Alldifferent Constraint: A Survey.” In: *Proceedings of the Sixth Annual Workshop of the ERCIM Working Group on Constraints*. 2001.
- [4] F. Rossi, P. van Beek, and T. Walsh, eds. *Handbook of Constraint Programming*. Elsevier, 2006. ISBN: 9780444527264.
- [5] F. Bacchus and A. Grove. “On the Forward Checking Algorithm.” In: *International Conference on Principles and Practice of Constraint Programming*. Berlin, Heidelberg: Springer Berlin Heidelberg, Sept. 1995, pp. 292–309.
- [6] J.-C. Régin. “A Filtering Algorithm for Constraints of Difference in CSPs.” In: *Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI-94)*. AAAI Conference on Artificial Intelligence, Volume 12. Seattle, Washington, USA, 1994.
- [7] U. catholique de Louvain. *Louv31x: Constraint Programming*. MOOC. LouvainX. URL: <https://www.edx.org/course/constraint-programming>.
- [8] C. Berge. “Two theorems in graph theory.” *Proceedings of the National Academy of Sciences*, **43**(9), 1957, pp. 842–844. DOI: [10.1073/pnas.43.9.842](https://doi.org/10.1073/pnas.43.9.842).
- [9] C. Berge. *Graphs and Hypergraphs*. North-Holland Mathematical Library. North-Holland, 1973. DOI: [10.1016/s0924-6509\(09\)x7013-3](https://doi.org/10.1016/s0924-6509(09)x7013-3). URL: [https://doi.org/10.1016/s0924-6509\(09\)x7013-3](https://doi.org/10.1016/s0924-6509(09)x7013-3).
- [10] J. E. Hopcroft, J. D. Ullman, and A. V. Aho. *Data Structures and Algorithms*. Vol. 175. Boston, MA, USA: Addison-Wesley, 1983.
- [11] M. Sharir. “A strong-connectivity algorithm and its applications in data flow analysis.” *Computers & Mathematics with Applications*, **7**(1), 1981, pp. 67–72.
- [12] R. Tarjan. “Depth-first search and linear graph algorithms.” *SIAM Journal on Computing*, **1**(2), 1972, pp. 146–160.
- [13] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. <http://www.deeplearningbook.org>. MIT Press, 2016.
- [14] I. H. Sarker. “Deep learning: a comprehensive overview on techniques, taxonomy, applications and research directions.” *SN Computer Science*, **2**(6), 2021, pp. 1–20.

- [15] T. Szandala. “Review and comparison of commonly used activation functions for deep neural networks.” In: *Bio-inspired neurocomputing*. Singapore: Springer Singapore, 2020, pp. 203–224.
- [16] O. Elharrouss, Y. Mahmood, Y. Bechqito, M. A. Serhani, E. Badidi, J. Riffi, and H. Tairi. “Loss Functions in Deep Learning: A Comprehensive Review.” *arXiv preprint arXiv:2504.04242*, (), 2025. arXiv: [2504.04242](#).
- [17] S. Ruder. “An overview of gradient descent optimization algorithms.” *arXiv preprint arXiv:1609.04747*, (), 2016. arXiv: [1609.04747](#).
- [18] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. “Dropout: a simple way to prevent neural networks from overfitting.” *The Journal of Machine Learning Research*, **15**(1), 2014, pp. 1929–1958.
- [19] W. L. Hamilton. *Graph representation learning*. Morgan & Claypool Publishers, 2020.
- [20] W. Hamilton, Z. Ying, and J. Leskovec. “Inductive representation learning on large graphs.” In: *Advances in Neural Information Processing Systems*. Vol. 30. 2017.
- [21] GeeksforGeeks. *Trade-offs between Exploration and Exploitation in Local Search Algorithms*. GeeksforGeeks, July 2025. URL: <https://www.geeksforgeeks.org/artificial-intelligence/trade-offs-between-exploration-and-exploitation-in-local-search-algorithms/> (visited on 09/20/2025).
- [22] F. Focacci, F. Laburthe, and A. Lodi. “Local Search and Constraint Programming.” In: *Handbook of Metaheuristics*. Boston, MA: Springer US, 2003, pp. 369–403.
- [23] D. Pisinger and S. Ropke. “Large Neighborhood Search.” In: *Handbook of Metaheuristics*. Cham: Springer International Publishing, 2018, pp. 99–127.
- [24] T. Huang, A. M. Ferber, A. Zharmagambetov, Y. Tian, and B. Dilkina. “Contrastive Predict-and-Search for Mixed Integer Linear Programs.” In: *Proceedings of the 41st International Conference on Machine Learning*. Ed. by R. Salakhutdinov, Z. Kolter, K. Heller, A. Weller, N. Oliver, J. Scarlett, and F. Berkenkamp. Vol. 235. Proceedings of Machine Learning Research. PMLR, July 2024, pp. 19757–19771. URL: <https://proceedings.mlr.press/v235/huang24f.html>.
- [25] C. Deruyck. *Prediction-based Propagation — Thesis Repository*. Master Thesis Repository. GitHub, 2025. URL: <https://github.com/corentinderuyck/Prediction-based-propagation---Thesis>.
- [26] aia-uclouvain. *MaxICP*. GitHub, 2025. URL: <https://github.com/aia-uclouvain/maxicp> (visited on 09/27/2025).
- [27] CRIL. *Problem Instances in XCSP3 Format*. 2025. URL: <https://xcsp.org/instances/>.
- [28] R. M. Haralick and G. L. Elliott. “Increasing Tree Search Efficiency for Constraint Satisfaction Problems.” *Artificial Intelligence*, **14**(3), July 1980, pp. 263–313. ISSN: 0004-3702. DOI: [10.1016/0004-3702\(80\)90006-4](https://doi.org/10.1016/0004-3702(80)90006-4).
- [29] H. Adel, F. Chen, and Y.-Y. Chen. “Ranking Convolutional Recurrent Neural Networks for Purchase Stage Identification on Imbalanced Twitter Data.” In: *Proceedings of the 15th Conference of the European Chapter of the Association for Computational Linguistics: Volume 2, Short Papers*. Apr. 2017, pp. 592–598.

- [30] D. M. W. Powers. *Evaluation: from precision, recall and F-measure to ROC, informedness, markedness and correlation*. arXiv preprint. 2020. arXiv: [2010.16061](https://arxiv.org/abs/2010.16061). URL: <https://arxiv.org/abs/2010.16061>.
- [31] O. Rainio, J. Teuho, and R. Klén. “Evaluation metrics and statistical tests for machine learning.” *Scientific Reports*, **14**(1), 2024, p. 6086.
- [32] *CECI: Consortium des Équipements de Calcul Intensif*. High Performance Computing Infrastructure. CECI – Universities of the Wallonia-Brussels Federation. URL: <https://www.ceci-hpc.be>.

Appendices

Instance	Nodes	Failures	Solutions	# AllDiff prop.
AllInterval-009	567158	283460	120	1448837
SchurrLemma-015-9-mod	725758	362880	0	18290066
SchurrLemma-020-9-mod	362878	181440	0	14159106
Kakuro-hard-010-sumdiff	22	11	1	375
Purdey	2	1	1	10
Kakuro-medium-129-sumdiff	294826	147413	1	3233337
KnightTour-06-ext03	406682	193480	9862	1241394
Queens-0009-m1	10906	5102	352	29617
Alpha	157720	78860	1	340401
SchurrLemma-012-9-mod	725758	362880	0	14335642
SchurrLemma-050-9-mod	362878	181440	0	63265266
Allergy	2	1	1	5
Kakuro-hard-056-sumdiff	44	22	1	643
SchurrLemma-030-9-mod	362878	181440	0	25115806
Queens-0012-m1	1578274	774938	14200	4390953
Kakuro-medium-035-sumdiff	308	154	1	3887
Kakuro-hard-062-sumdiff	72	36	1	1195
Kakuro-medium-062-sumdiff	64	32	1	953
AllInterval-006	1110	532	24	2704
Kakuro-hard-137-sumdiff	739460	369730	1	9699484
Kakuro-medium-086-sumdiff	114	57	1	2072
SchurrLemma-100-9-mod	362878	181440	0	227947666

Table 11.1: Number of nodes, failures, solutions, and calls to the propagate method for each training instance

Instance	Nodes	Failures	Solutions	# AllDiff prop.
bqwh-15-106-51 _X 2	462	39	193	10705
ColouredQueens-07	2771998	1365840	20160	100035595
Sudoku-s12a-alldiff	2	1	1	225
Sudoku-s11c-alldiff	2	1	1	117
bqwh-18-141-50 _X 2	618	122	188	18959
Subisomorphism-si2-m4Dr6-m625-04	9504	145	4608	18194
Langford-4-09	114	58	0	209
ColouredQueens-03	4	3	0	36
Sudoku-s13a-alldiff	2	1	1	89
Sudoku-s08c-alldiff	2	1	1	178
Subisomorphism-g16-g46	2278	1140	0	4313
bqwh-15-106-02 _X 2	102	23	29	2618
sudoku368 _X 2	4	2	1	1004
Langford-2-06	122	62	0	205
Sudoku-s11b-alldiff	2	1	1	145
bqwh-15-106-07 _X 2	332	75	92	9055
Sudoku-s03b-alldiff	2	1	1	136
OpenStacks-m1-wbop-30-10-1	202696	101349	0	339948
bqwh-18-141-75 _X 2	514	225	33	19998
MagicSquare-03-sum	62	24	8	153
Ortholatin-005	10942	5040	432	171795
bqwh-15-106-81 _X 2	164	48	35	4314
bqwh-15-106-25 _X 2	604	224	79	17848
Sudoku-s13c-alldiff	2	1	1	83
Subisomorphism-si4-m4Dr2-m625-02	100	3	48	204
CostasArray-10	176932	86307	2160	2657537

Table 11.2: Number of nodes, failures, solutions and calls to the propagate method for each test instance

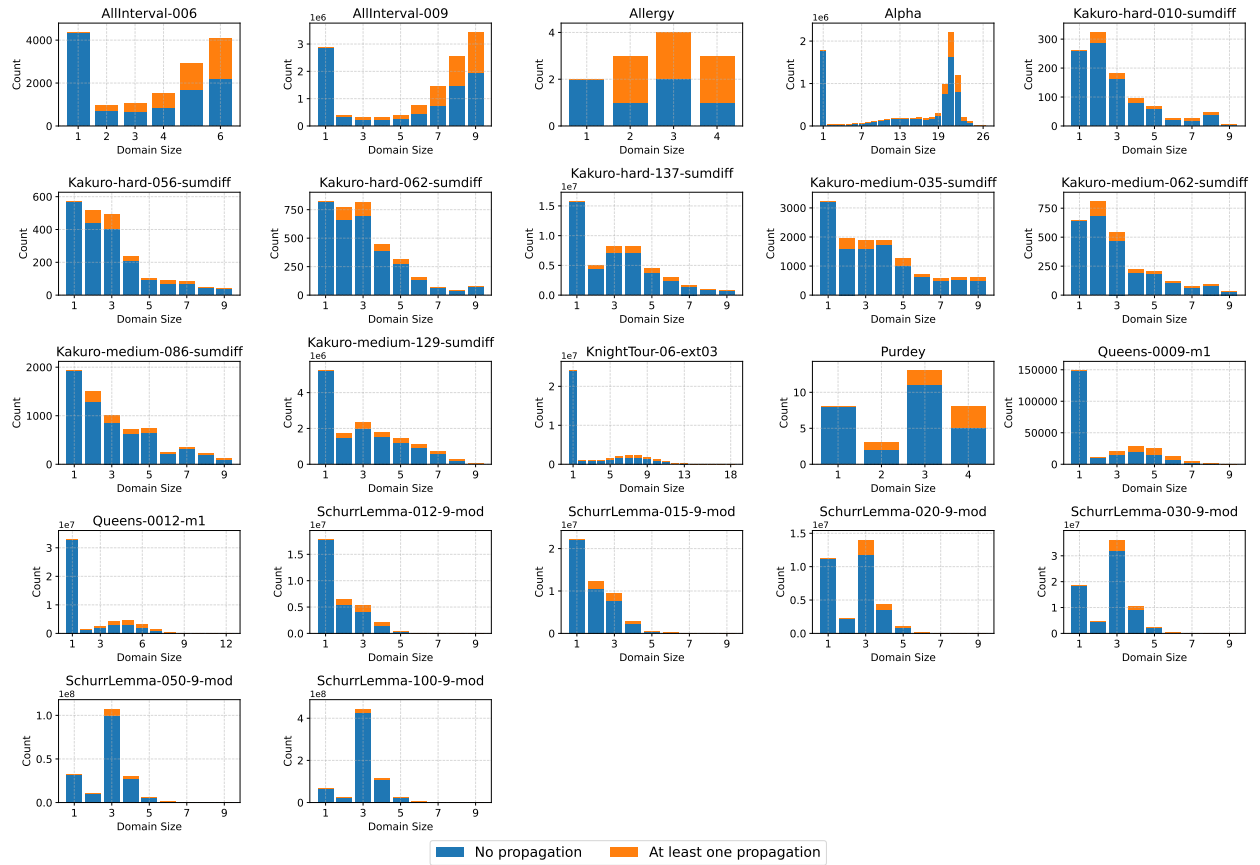


Figure 11.1: Histograms of the training instances illustrating the domain sizes observed during propagation, distinguishing domains that were reduced from those that were not

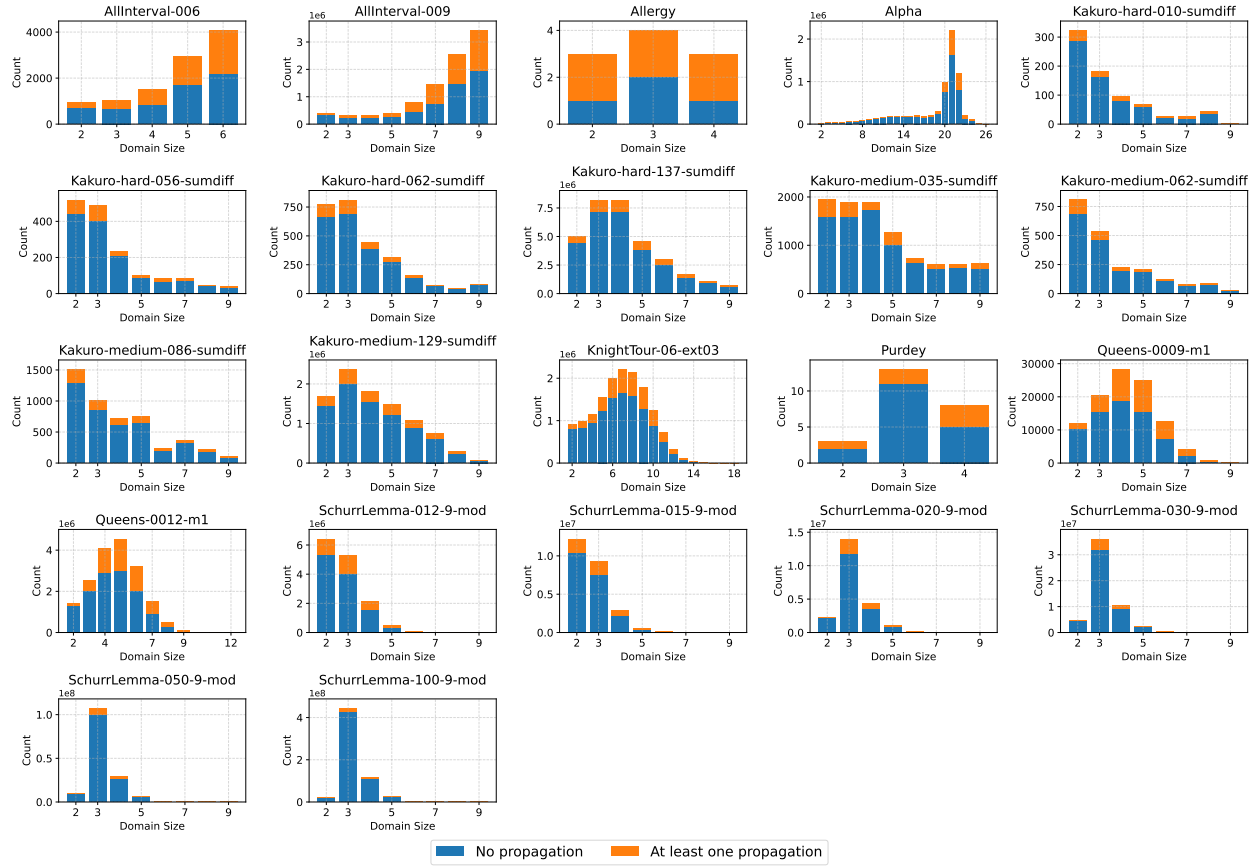


Figure 11.2: Histograms of the training instances illustrating the domain sizes observed during propagation, distinguishing domains that were reduced from those that were not, with single-value domains removed

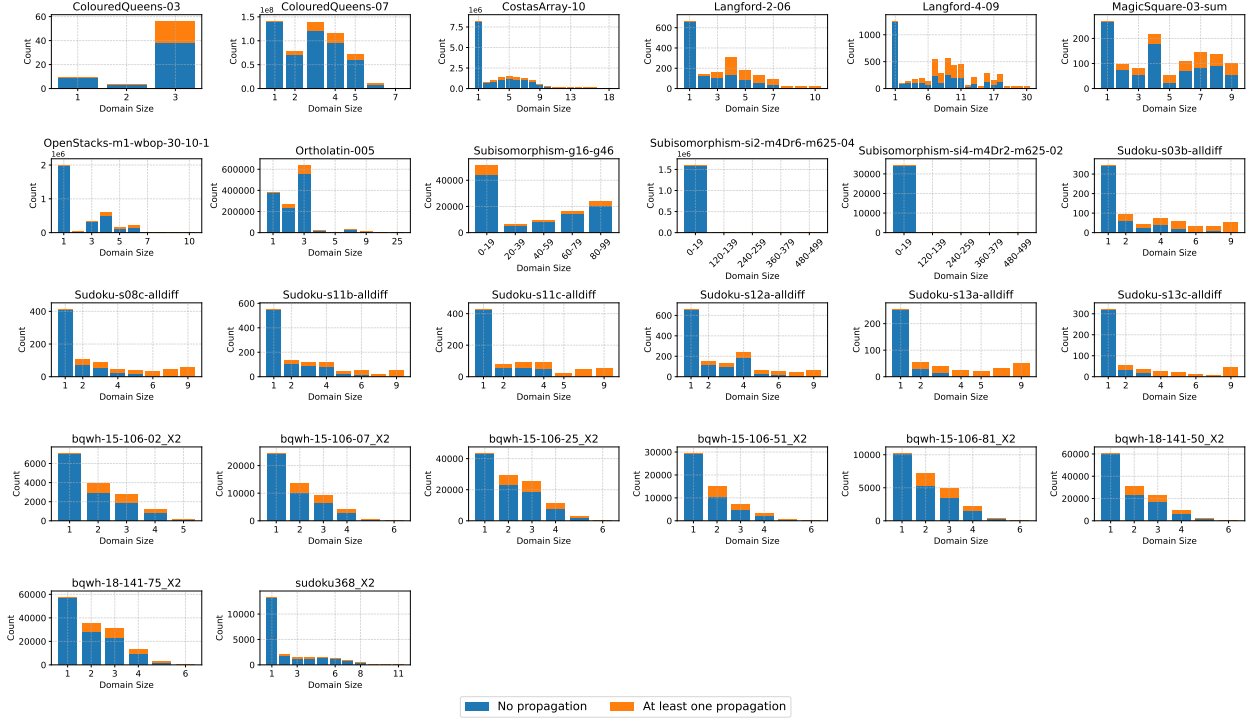


Figure 11.3: Histograms of the test instances illustrating the domain sizes observed during propagation, distinguishing domains that were reduced from those that were not

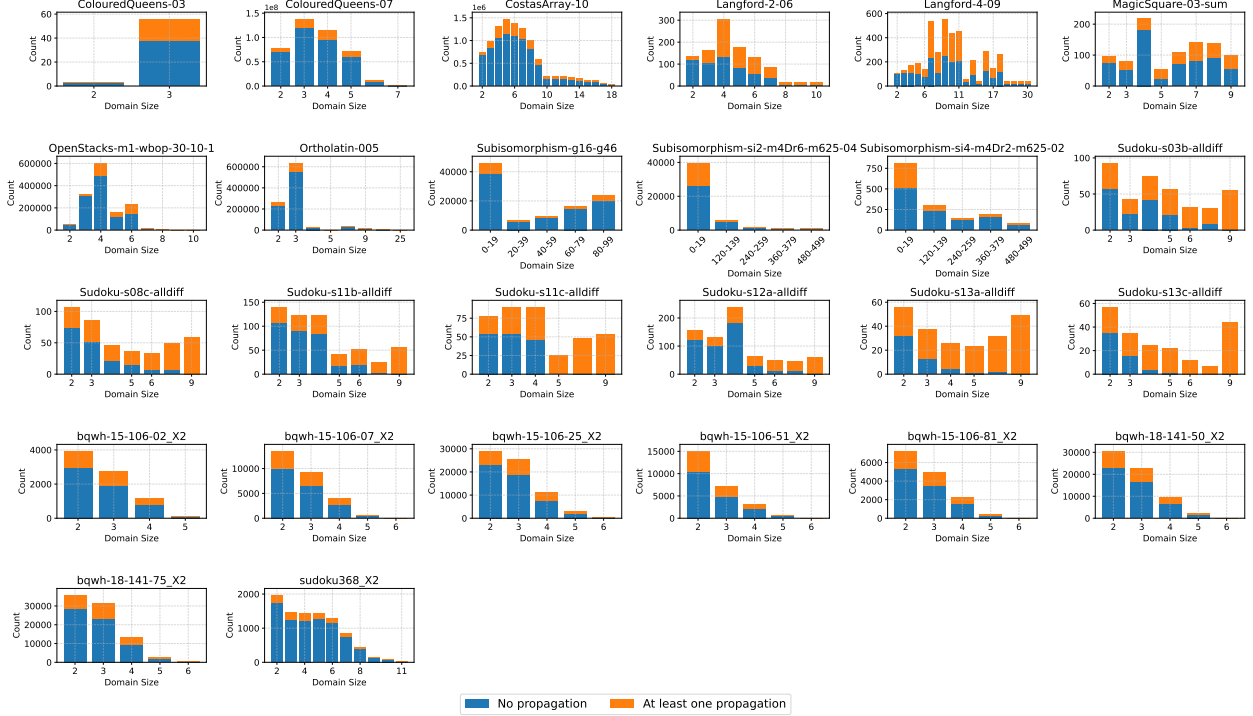


Figure 11.4: Histograms of the test instances illustrating the domain sizes observed during propagation, distinguishing domains that were reduced from those that were not, with single-value domains removed

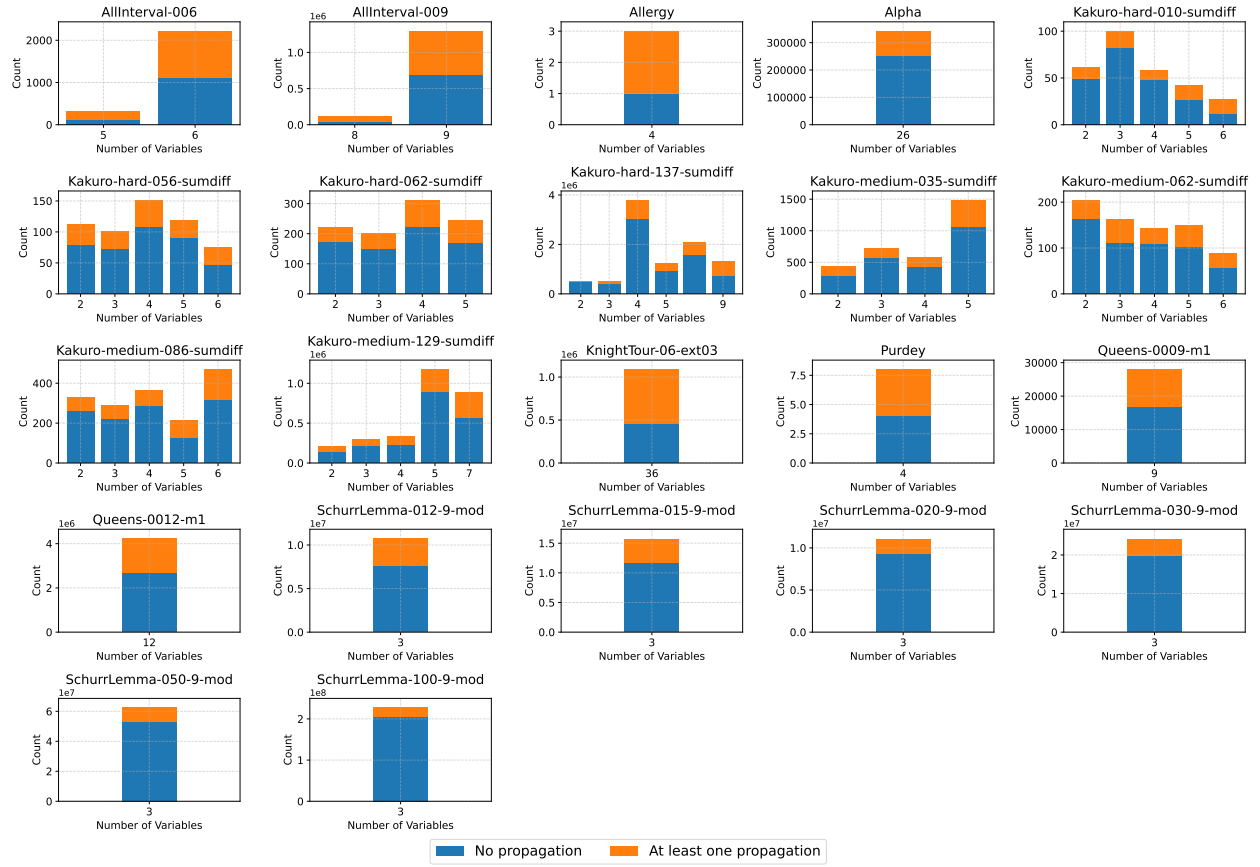


Figure 11.5: Histograms of the training instances illustrating the number of variables linked by an AllDifferent constraint observed during propagation, distinguishing domains that were reduced from those that were not

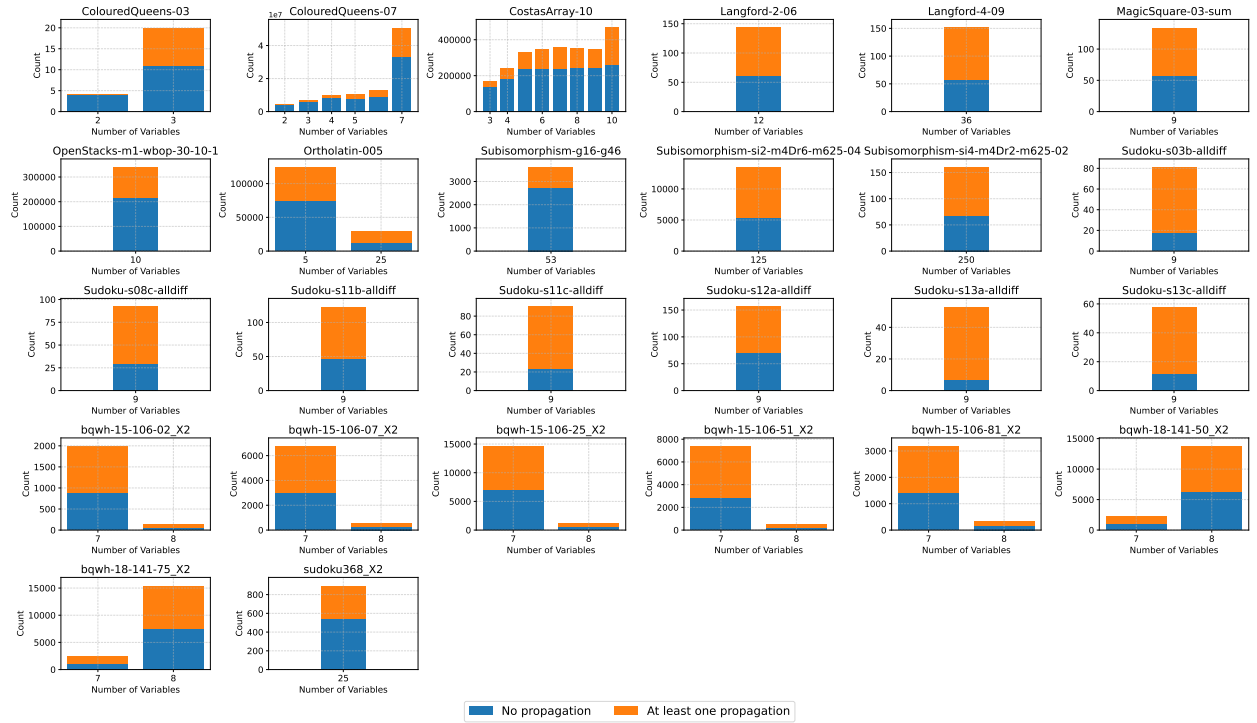


Figure 11.6: Histograms of the test instances illustrating the number of variables linked by an AllDifferent constraint observed during propagation, distinguishing domains that were reduced from those that were not

Table 11.3: Table summarizing all metrics for each training instance computed with the baseline model

File	Acc. global	Prec. 0	Recall 0	F1 0	Support 0	Prec. 1	Recall 1	F1 1	Support 1
AllInterval-006	0.9988	0.9996	0.9991	0.9993	49605	0.9912	0.9959	0.9935	5076
AllInterval-009	0.9970	0.9982	0.9986	0.9984	69539083	0.9782	0.9721	0.9751	4491537
Allergy	1.0000	1.0000	1.0000	1.0000	20	1.0000	1.0000	1.0000	12
Alpha	0.9980	0.9998	0.9982	0.9990	132012279	0.8751	0.9860	0.9272	1720383
Kakuro-hard-010-sumdiff	0.9883	0.9996	0.9882	0.9939	2804	0.7692	0.9910	0.8661	111
Kakuro-hard-056-sumdiff	0.9715	0.9995	0.9708	0.9849	6160	0.5862	0.9884	0.7359	258
Kakuro-hard-062-sumdiff	0.9863	0.9993	0.9865	0.9928	10272	0.7342	0.9821	0.8403	391
Kakuro-hard-137-sumdiff	0.9848	0.9991	0.9852	0.9921	146501127	0.6844	0.9724	0.8033	4827578
Kakuro-medium-035-sumdiff	0.9912	0.9997	0.9912	0.9954	44343	0.7924	0.9900	0.8803	1500
Kakuro-medium-062-sumdiff	0.9699	0.9979	0.9707	0.9841	7806	0.5751	0.9509	0.7168	326
Kakuro-medium-086-sumdiff	0.9829	0.9987	0.9836	0.9911	20684	0.6858	0.9649	0.8017	769
Kakuro-medium-129-sumdiff	0.9823	0.9992	0.9824	0.9907	43473903	0.6754	0.9800	0.7997	1624282
KnightTour-06-ext03	0.9977	0.9992	0.9984	0.9988	124103987	0.9537	0.9755	0.9645	4073809
Purdey	1.0000	1.0000	1.0000	1.0000	75	1.0000	1.0000	1.0000	10
Queens-0009-m1	0.9998	0.9999	0.9999	0.9999	548012	0.9986	0.9986	0.9986	35186
Queens-0012-m1	0.9996	0.9998	0.9998	0.9998	111655796	0.9956	0.9969	0.9962	5702335
SchurrLemma-012-9-mod	1.0000	1.0000	1.0000	1.0000	55031201	1.0000	1.0000	1.0000	3144267
SchurrLemma-015-9-mod	1.0000	1.0000	1.0000	1.0000	85749044	1.0000	1.0000	1.0000	4485654
SchurrLemma-020-9-mod	1.0000	1.0000	1.0000	1.0000	79308261	1.0000	1.0000	1.0000	3309219
SchurrLemma-030-9-mod	1.0000	1.0000	1.0000	1.0000	187707401	1.0000	1.0000	1.0000	6097149
SchurrLemma-050-9-mod	1.0000	1.0000	1.0000	1.0000	521797731	1.0000	1.0000	1.0000	11310129
SchurrLemma-100-9-mod	1.0000	1.0000	1.0000	1.0000	2041227181	1.0000	1.0000	1.0000	24342579

Table 11.4: Table summarizing all metrics for each test instance computed with the baseline model

File	Acc. global	Prec. 0	Recall 0	F1 0	Support 0	Prec. 1	Recall 1	F1 1	Support 1
ColouredQueens-03	1.0000	1.0000	1.0000	1.0000	165	1.0000	1.0000	1.0000	18
ColouredQueens-07	0.9750	0.9942	0.9795	0.9868	1519939348	0.6760	0.8822	0.7654	73602480
CostasArray-10	0.9907	0.9992	0.9912	0.9952	71519494	0.7661	0.9733	0.8574	2106757
Langford-2-06	0.8542	0.8881	0.9391	0.9129	4402	0.6460	0.4842	0.5535	1010
Langford-4-09	0.6123	0.7951	0.6510	0.7159	34024	0.3212	0.4959	0.3899	11328
MagicSquare-03-sum	0.9815	0.9946	0.9852	0.9899	5068	0.8434	0.9374	0.8879	431
OpenStacks-m1-wbop-30-10-1	0.9993	0.9998	0.9994	0.9996	7579019	0.9871	0.9964	0.9917	337802
Ortholatin-005	0.9801	0.9795	1.0000	0.9897	3120613	1.0000	0.5754	0.7305	153601
Subisomorphism-g16-g46	0.9309	0.9990	0.9314	0.9640	14358706	0.0702	0.8464	0.1297	87899
Subisomorphism-si2-m4Dr6-m625-04	0.7032	0.9969	0.7041	0.8253	17591559	0.0081	0.5226	0.0159	80909
Subisomorphism-si4-m4Dr2-m625-02	0.6876	0.9939	0.6879	0.8130	1252654	0.0262	0.6654	0.0505	15834
Sudoku-s03b-alldiff	0.9524	0.9711	0.9688	0.9700	1701	0.8806	0.8886	0.8846	440
Sudoku-s08c-alldiff	0.9596	0.9651	0.9849	0.9749	1852	0.9355	0.8602	0.8962	472
Sudoku-s11b-alldiff	0.9481	0.9864	0.9519	0.9688	2432	0.7771	0.9273	0.8456	440
Sudoku-s11c-alldiff	0.9691	0.9873	0.9738	0.9805	1681	0.9016	0.9505	0.9254	424
Sudoku-s12a-alldiff	0.9240	0.9906	0.9219	0.9550	3329	0.6302	0.9386	0.7540	472
Sudoku-s13a-alldiff	0.9497	0.9628	0.9659	0.9644	939	0.9177	0.9107	0.9142	392
Sudoku-s13c-alldiff	0.9803	0.9826	0.9902	0.9864	914	0.9739	0.9545	0.9641	352
bqwh-15-106-02_X2	0.9390	0.9725	0.9594	0.9659	25742	0.6733	0.7553	0.7119	2852
bqwh-15-106-07_X2	0.9357	0.9766	0.9516	0.9639	89078	0.6359	0.7879	0.7038	9564
bqwh-15-106-25_X2	0.9059	0.9775	0.9170	0.9463	216852	0.5056	0.8009	0.6199	22971
bqwh-15-106-51_X2	0.9173	0.9739	0.9330	0.9530	87038	0.5686	0.7791	0.6574	9866
bqwh-15-106-81_X2	0.9050	0.9679	0.9248	0.9459	45132	0.5278	0.7329	0.6137	5177
bqwh-18-141-50_X2	0.9305	0.9762	0.9467	0.9612	217515	0.5878	0.7671	0.6656	21547
bqwh-18-141-75_X2	0.9111	0.9720	0.9288	0.9499	265270	0.5114	0.7359	0.6035	26850
sudoku368_X2	0.8431	0.9931	0.8442	0.9126	52489	0.1350	0.8050	0.2312	1585

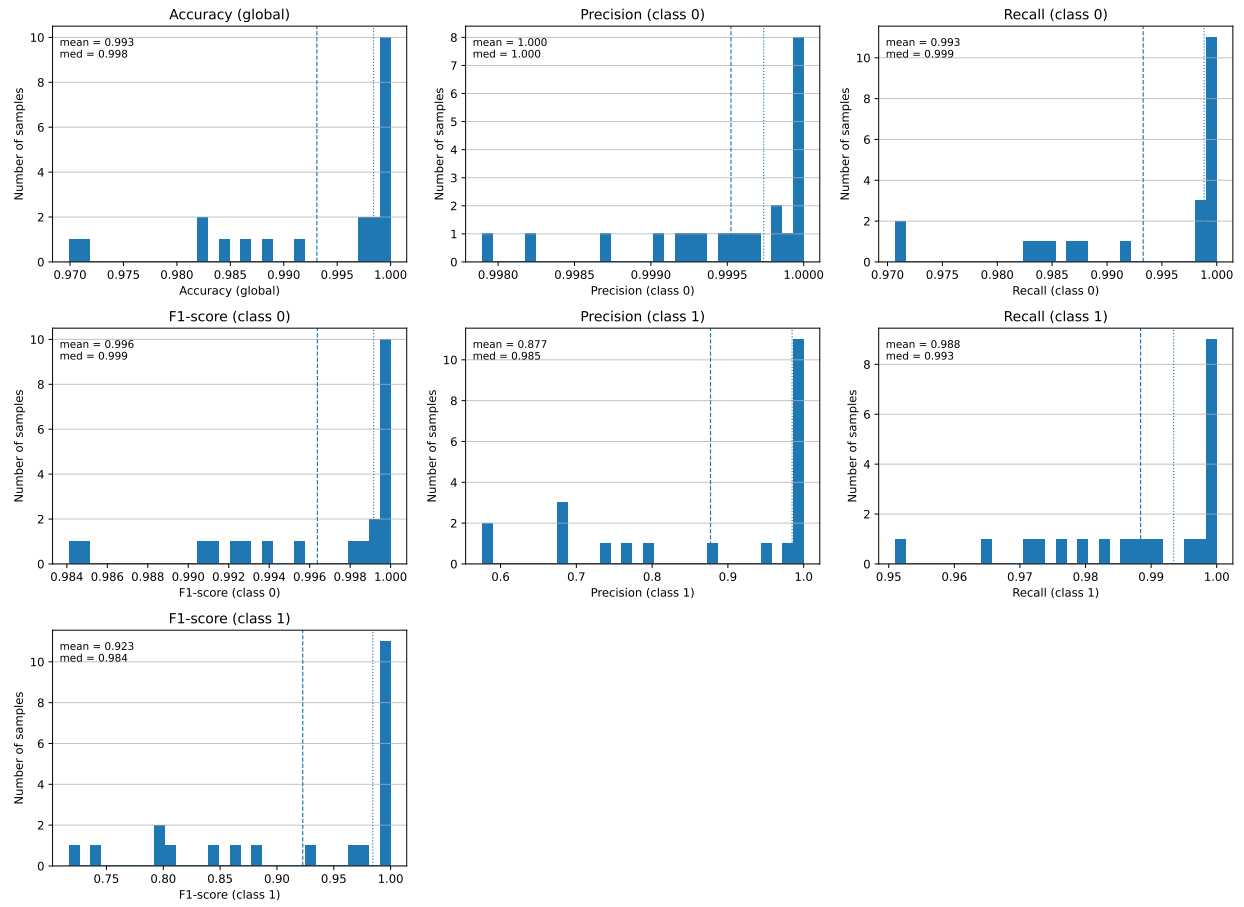


Figure 11.7: Histogram of the evaluation metrics of the baseline model on the training data

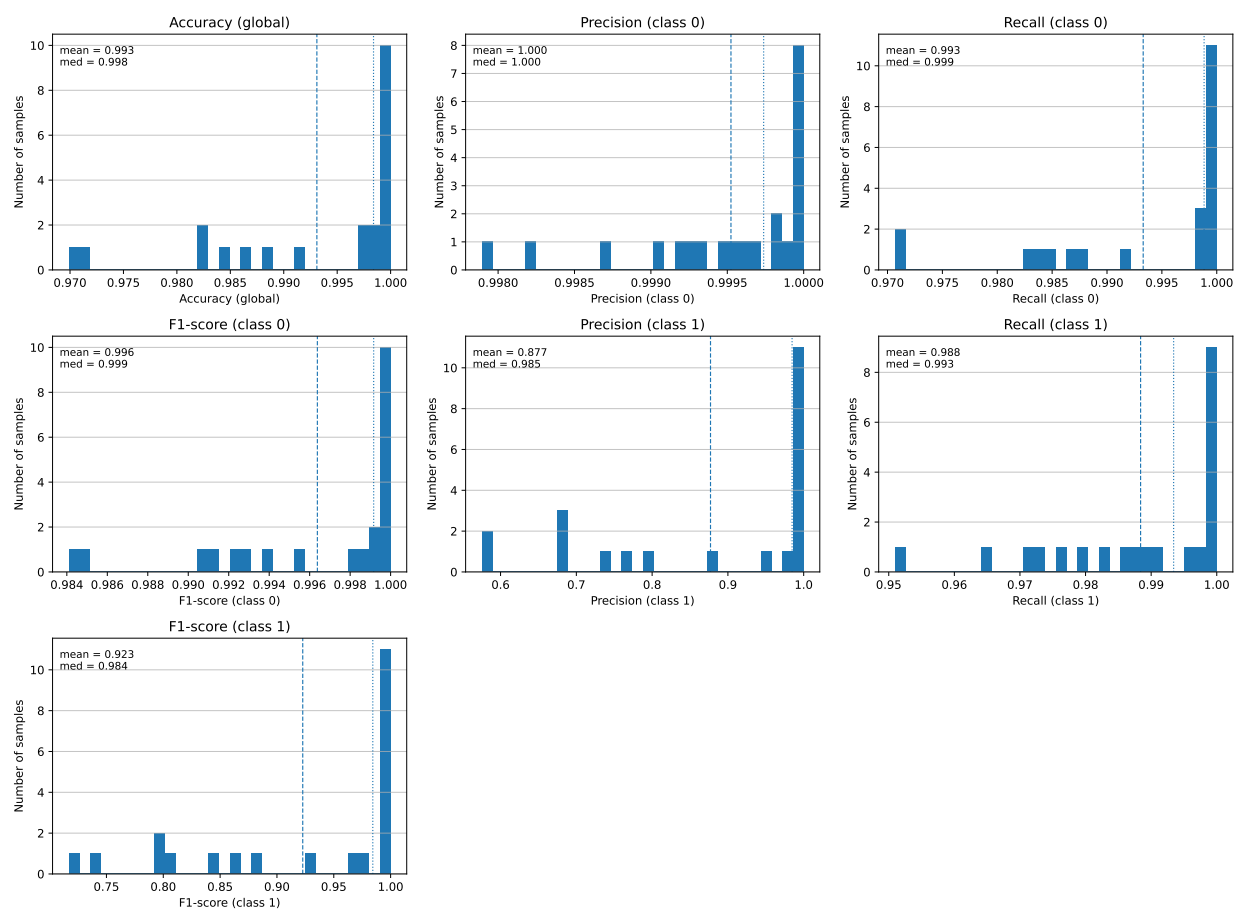


Figure 11.8: Histogram of the evaluation metrics of the baseline model on the test data

Table 11.5: Table summarizing all metrics for each training instance computed with the small model

File	Acc. global	Prec. 0	Recall 0	F1 0	Support 0	Prec. 1	Recall 1	F1 1	Support 1
AllInterval-006	0.9936	0.9930	1.0000	0.9965	49605	1.0000	0.9314	0.9645	5076
AllInterval-009	0.9975	0.9979	0.9995	0.9987	69539083	0.9921	0.9668	0.9793	4491537
Allergy	0.8125	0.7692	1.0000	0.8696	20	1.0000	0.5000	0.6667	12
Alpha	0.9952	0.9998	0.9953	0.9976	132012279	0.7344	0.9879	0.8425	1720383
Kakuro-hard-010-sumdiff	0.9774	0.9960	0.9804	0.9881	2804	0.6452	0.9009	0.7519	111
Kakuro-hard-056-sumdiff	0.9590	0.9950	0.9622	0.9783	6160	0.4946	0.8837	0.6342	258
Kakuro-hard-062-sumdiff	0.9829	0.9971	0.9851	0.9911	10272	0.7029	0.9258	0.7991	391
Kakuro-hard-137-sumdiff	0.9786	0.9943	0.9836	0.9889	146501127	0.6241	0.8282	0.7118	4827578
Kakuro-medium-035-sumdiff	0.9805	0.9975	0.9823	0.9899	44343	0.6397	0.9267	0.7569	1500
Kakuro-medium-062-sumdiff	0.9555	0.9914	0.9620	0.9765	7806	0.4677	0.8006	0.5905	326
Kakuro-medium-086-sumdiff	0.9704	0.9955	0.9737	0.9845	20684	0.5549	0.8804	0.6807	769
Kakuro-medium-129-sumdiff	0.9689	0.9948	0.9728	0.9837	43473903	0.5431	0.8639	0.6669	1624282
KnightTour-06-ext03	0.9965	0.9970	0.9995	0.9982	124103987	0.9828	0.9073	0.9435	4073809
Purdey	0.9529	0.9494	1.0000	0.9740	75	1.0000	0.6000	0.7500	10
Queens-0009-m1	0.9960	0.9958	1.0000	0.9979	548012	1.0000	0.9341	0.9659	35186
Queens-0012-m1	0.9972	0.9971	1.0000	0.9985	111655796	0.9991	0.9434	0.9704	5702335
SchurrLemma-012-9-mod	0.9999	1.0000	0.9999	0.9999	55031201	0.9974	0.9998	0.9986	3144267
SchurrLemma-015-9-mod	0.9998	1.0000	0.9998	0.9999	85749044	0.9969	0.9998	0.9983	4485654
SchurrLemma-020-9-mod	0.9982	0.9985	0.9997	0.9991	79308261	0.9925	0.9631	0.9776	3309219
SchurrLemma-030-9-mod	0.9876	0.9876	0.9998	0.9936	187707401	0.9883	0.6128	0.7565	6097149
SchurrLemma-050-9-mod	0.9863	0.9864	0.9998	0.9931	521797731	0.9798	0.3633	0.5301	11310129
SchurrLemma-100-9-mod	0.9906	0.9906	0.9999	0.9952	2041227181	0.9645	0.2071	0.3410	24342579

Table 11.6: Table summarizing all metrics for each test instance computed with the small model

File	Acc. global	Prec. 0	Recall 0	F1 0	Support 0	Prec. 1	Recall 1	F1 1	Support 1
ColouredQueens-03	1.0000	1.0000	1.0000	1.0000	165	1.0000	1.0000	1.0000	18
ColouredQueens-07	0.9744	0.9797	0.9937	0.9867	1519939348	0.8161	0.5748	0.6745	73602480
CostasArray-10	0.9882	0.9953	0.9925	0.9939	71519494	0.7681	0.8410	0.8029	2106757
Langford-2-06	0.8777	0.8697	0.9993	0.9300	4402	0.9915	0.3475	0.5147	1010
Langford-4-09	0.7305	0.8071	0.8420	0.8242	34024	0.4545	0.3955	0.4230	11328
MagicSquare-03-sum	0.9665	0.9886	0.9749	0.9817	5068	0.7465	0.8677	0.8026	431
OpenStacks-m1-wbop-30-10-1	0.9856	0.9855	0.9997	0.9925	7579019	0.9895	0.6704	0.7992	337802
Ortholatin-005	0.9601	0.9599	1.0000	0.9795	3120613	1.0000	0.1504	0.2614	153601
Subisomorphism-g16-g46	0.9164	0.9996	0.9163	0.9561	14358706	0.0640	0.9349	0.1198	87899
Subisomorphism-si2-m4Dr6-m625-04	0.9483	0.9971	0.9508	0.9734	17591559	0.0353	0.3908	0.0647	80909
Subisomorphism-si4-m4Dr2-m625-02	0.7119	0.9954	0.7116	0.8299	1252654	0.0313	0.7371	0.0600	15834
Sudoku-s03b-alldiff	0.9393	0.9290	1.0000	0.9632	1701	1.0000	0.7045	0.8267	440
Sudoku-s08c-alldiff	0.9299	0.9191	1.0000	0.9578	1852	1.0000	0.6547	0.7913	472
Sudoku-s11b-alldiff	0.9516	0.9466	0.9992	0.9722	2432	0.9934	0.6886	0.8134	440
Sudoku-s11c-alldiff	0.9344	0.9241	1.0000	0.9606	1681	1.0000	0.6745	0.8056	424
Sudoku-s12a-alldiff	0.9595	0.9592	0.9961	0.9773	3329	0.9622	0.7013	0.8113	472
Sudoku-s13a-alldiff	0.9181	0.8960	1.0000	0.9451	939	1.0000	0.7219	0.8385	392
Sudoku-s13c-alldiff	0.9250	0.9058	1.0000	0.9506	914	1.0000	0.7301	0.8440	352
bqwh-15-106-02_X2	0.9353	0.9330	1.0000	0.9653	25742	1.0000	0.3517	0.5204	2852
bqwh-15-106-07_X2	0.9384	0.9365	0.9996	0.9670	89078	0.9907	0.3683	0.5369	9564
bqwh-15-106-25_X2	0.9391	0.9376	0.9991	0.9674	216852	0.9780	0.3724	0.5394	22971
bqwh-15-106-51_X2	0.9363	0.9348	0.9988	0.9657	87038	0.9738	0.3850	0.5518	9866
bqwh-15-106-81_X2	0.9295	0.9275	0.9995	0.9622	45132	0.9863	0.3191	0.4822	5177
bqwh-18-141-50_X2	0.9427	0.9409	0.9998	0.9695	217515	0.9957	0.3663	0.5355	21547
bqwh-18-141-75_X2	0.9360	0.9351	0.9989	0.9659	265270	0.9662	0.3151	0.4752	26850
sudoku368_X2	0.9514	0.9834	0.9663	0.9748	52489	0.2914	0.4587	0.3564	1585

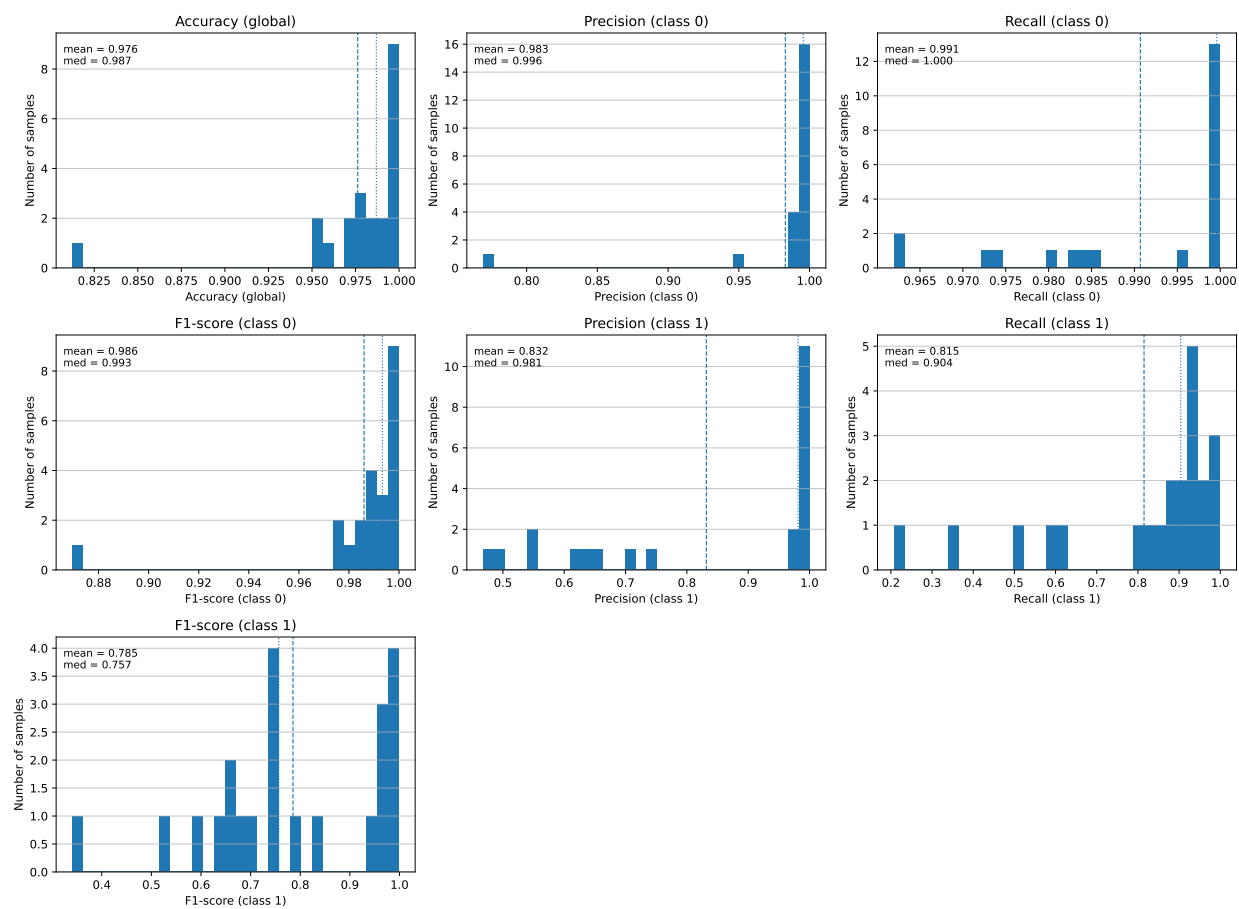


Figure 11.9: Histogram of the evaluation metrics of the small model on the training data

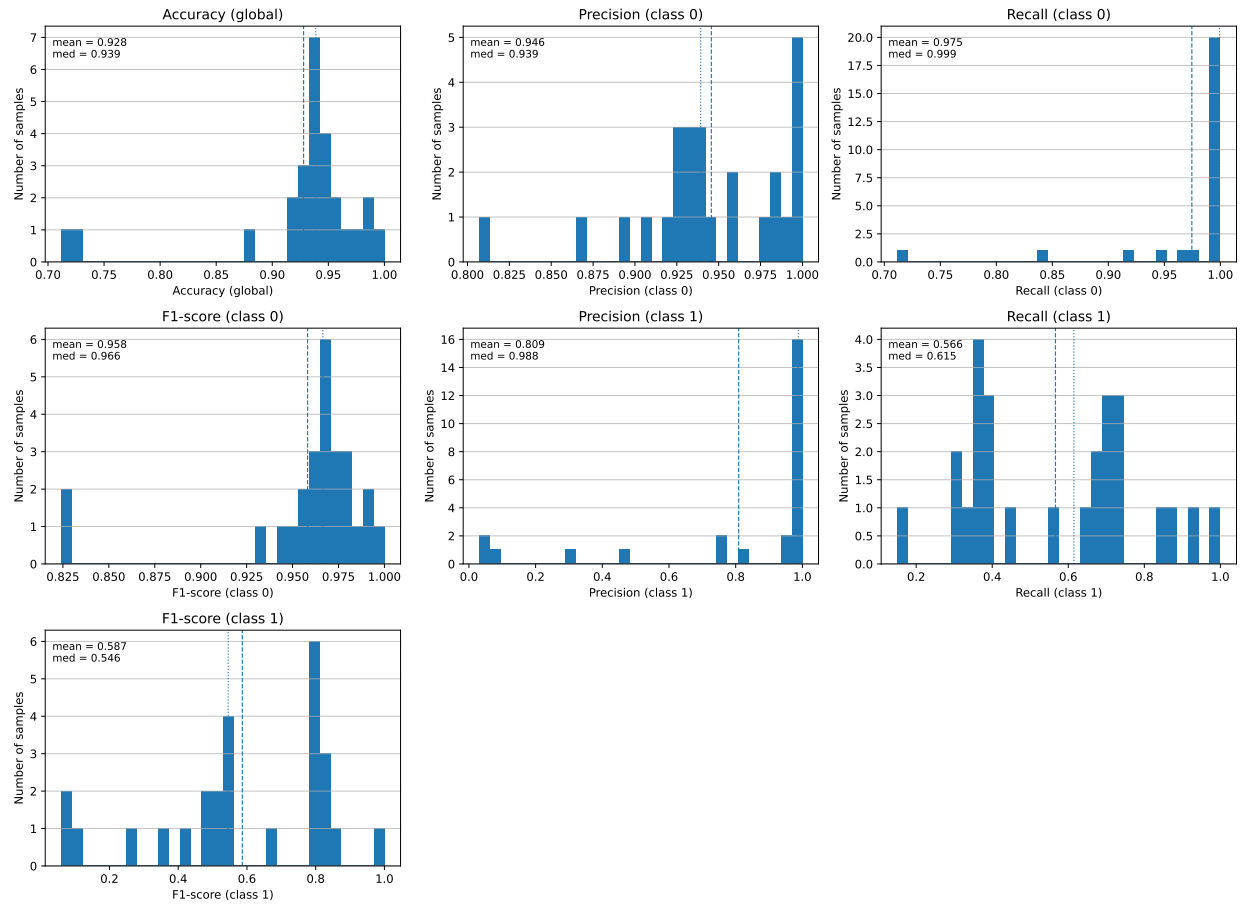


Figure 11.10: Histogram of the evaluation metrics of the small model on the test data

Table 11.7: Table summarizing all metrics for each training instance computed with the random model following the distribution of the baseline model

File	Acc. global	Prec. 0	Recall 0	F1 0	Support 0	Prec. 1	Recall 1	F1 1	Support 1
AllInterval-006	0.9017	0.9074	0.9930	0.9483	49605	0.1221	0.0095	0.0176	5076
AllInterval-009	0.9332	0.9393	0.9931	0.9654	69539083	0.0605	0.0069	0.0124	4491537
Allergy	0.6250	0.6250	1.0000	0.7692	20	0.0000	0.0000	0.0000	12
Alpha	0.9804	0.9871	0.9931	0.9901	132012279	0.0129	0.0070	0.0091	1720383
Kakuro-hard-010-sumdiff	0.9578	0.9621	0.9954	0.9784	2804	0.0714	0.0090	0.0160	111
Kakuro-hard-056-sumdiff	0.9526	0.9597	0.9924	0.9757	6160	0.0208	0.0039	0.0065	258
Kakuro-hard-062-sumdiff	0.9558	0.9632	0.9920	0.9774	10272	0.0238	0.0051	0.0084	391
Kakuro-hard-137-sumdiff	0.9616	0.9681	0.9931	0.9804	146501127	0.0317	0.0069	0.0113	4827578
Kakuro-medium-035-sumdiff	0.9610	0.9674	0.9932	0.9801	44343	0.0443	0.0093	0.0154	1500
Kakuro-medium-062-sumdiff	0.9524	0.9600	0.9918	0.9756	7806	0.0448	0.0092	0.0153	326
Kakuro-medium-086-sumdiff	0.9573	0.9640	0.9927	0.9782	20684	0.0132	0.0026	0.0043	769
Kakuro-medium-129-sumdiff	0.9575	0.9640	0.9930	0.9783	43473903	0.0359	0.0069	0.0116	1624282
KnightTour-06-ext03	0.9617	0.9682	0.9931	0.9805	124103987	0.0319	0.0070	0.0114	4073809
Purdey	0.8706	0.8810	0.9867	0.9308	75	0.0000	0.0000	0.0000	10
Queens-0009-m1	0.9337	0.9397	0.9932	0.9657	548012	0.0609	0.0068	0.0123	35186
Queens-0012-m1	0.9451	0.9514	0.9930	0.9718	111655796	0.0487	0.0070	0.0122	5702335
SchurrLemma-012-9-mod	0.9397	0.9460	0.9930	0.9689	55031201	0.0545	0.0070	0.0125	3144267
SchurrLemma-015-9-mod	0.9440	0.9503	0.9931	0.9712	85749044	0.0498	0.0069	0.0122	4485654
SchurrLemma-020-9-mod	0.9536	0.9599	0.9931	0.9762	79308261	0.0398	0.0069	0.0118	3309219
SchurrLemma-030-9-mod	0.9620	0.9685	0.9930	0.9806	187707401	0.0313	0.0069	0.0113	6097149
SchurrLemma-050-9-mod	0.9721	0.9788	0.9930	0.9859	521797731	0.0211	0.0069	0.0104	11310129
SchurrLemma-100-9-mod	0.9814	0.9882	0.9931	0.9906	2041227181	0.0118	0.0070	0.0088	24342579

Table 11.8: Table summarizing all metrics for each test instance computed with the random model following the distribution of the baseline model

File	Acc. global	Prec. 0	Recall 0	F1 0	Support 0	Prec. 1	Recall 1	F1 1	Support 1
ColouredQueens-03	0.8907	0.9006	0.9879	0.9422	165	0.0000	0.0000	0.0000	18
ColouredQueens-07	0.9475	0.9538	0.9930	0.9730	1519939348	0.0461	0.0069	0.0121	73602480
CostasArray-10	0.9648	0.9714	0.9931	0.9821	71519494	0.0284	0.0069	0.0111	2106757
Langford-2-06	0.8108	0.8139	0.9948	0.8953	4402	0.2812	0.0089	0.0173	1010
Langford-4-09	0.7469	0.7503	0.9930	0.8548	34024	0.2663	0.0076	0.0148	11328
MagicSquare-03-sum	0.9147	0.9215	0.9919	0.9554	5068	0.0682	0.0070	0.0126	431
OpenStacks-m1-wbop-30-10-1	0.9510	0.9573	0.9931	0.9749	7579019	0.0422	0.0069	0.0118	337802
Ortholatin-005	0.9468	0.9531	0.9931	0.9727	3120613	0.0469	0.0069	0.0120	153601
Subisomorphism-g16-g46	0.9870	0.9939	0.9930	0.9935	14358706	0.0059	0.0067	0.0063	87899
Subisomorphism-si2-m4Dr6-m625-04	0.9886	0.9954	0.9931	0.9942	17591559	0.0048	0.0073	0.0058	80909
Subisomorphism-si4-m4Dr2-m625-02	0.9809	0.9875	0.9932	0.9903	1252654	0.0120	0.0066	0.0085	15834
Sudoku-s03b-alldiff	0.7875	0.7933	0.9906	0.8810	1701	0.0588	0.0023	0.0044	440
Sudoku-s08c-alldiff	0.7909	0.7959	0.9919	0.8832	1852	0.0625	0.0021	0.0041	472
Sudoku-s11b-alldiff	0.8409	0.8461	0.9926	0.9135	2432	0.0526	0.0023	0.0044	440
Sudoku-s11c-alldiff	0.7957	0.7991	0.9941	0.8860	1681	0.2857	0.0094	0.0183	424
Sudoku-s12a-alldiff	0.8708	0.8760	0.9931	0.9309	3329	0.1481	0.0085	0.0160	472
Sudoku-s13a-alldiff	0.7032	0.7057	0.9936	0.8253	939	0.3333	0.0077	0.0150	392
Sudoku-s13c-alldiff	0.7188	0.7214	0.9945	0.8362	914	0.1667	0.0028	0.0056	352
bqwh-15-106-02_X2	0.8945	0.9003	0.9928	0.9443	25742	0.1014	0.0074	0.0137	2852
bqwh-15-106-07_X2	0.8974	0.9031	0.9930	0.9459	89078	0.1068	0.0078	0.0146	9564
bqwh-15-106-25_X2	0.8986	0.9042	0.9931	0.9466	216852	0.0943	0.0068	0.0127	22971
bqwh-15-106-51_X2	0.8927	0.8982	0.9931	0.9433	87038	0.1028	0.0070	0.0131	9866
bqwh-15-106-81_X2	0.8914	0.8970	0.9929	0.9425	45132	0.0886	0.0060	0.0112	5177
bqwh-18-141-50_X2	0.9040	0.9098	0.9928	0.9495	217515	0.0851	0.0067	0.0125	21547
bqwh-18-141-75_X2	0.9025	0.9081	0.9931	0.9487	265270	0.0976	0.0074	0.0137	26850
sudoku368_X2	0.9635	0.9706	0.9924	0.9814	52489	0.0197	0.0050	0.0080	1585

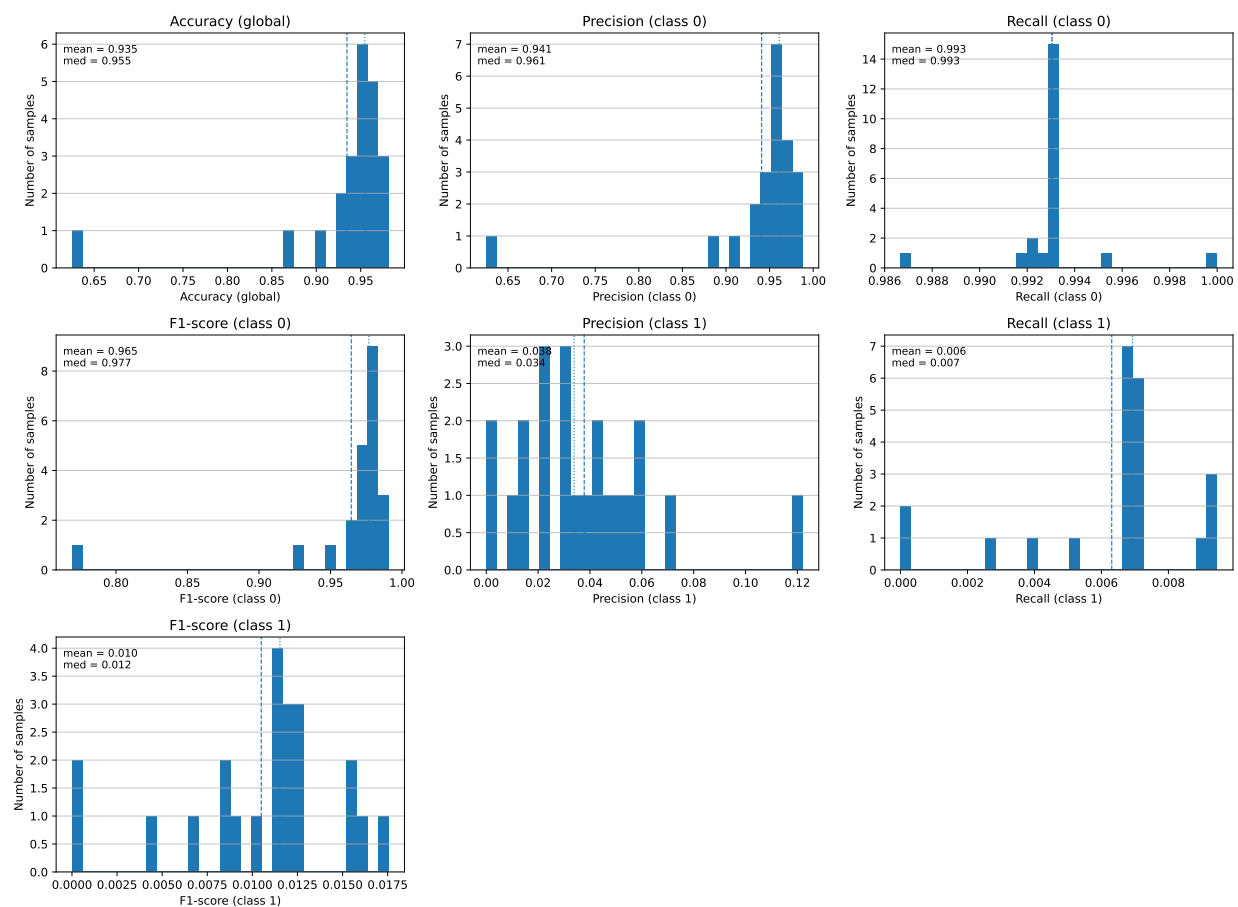


Figure 11.11: Histogram of the evaluation metrics of the random model following the baseline model's distribution on the training data

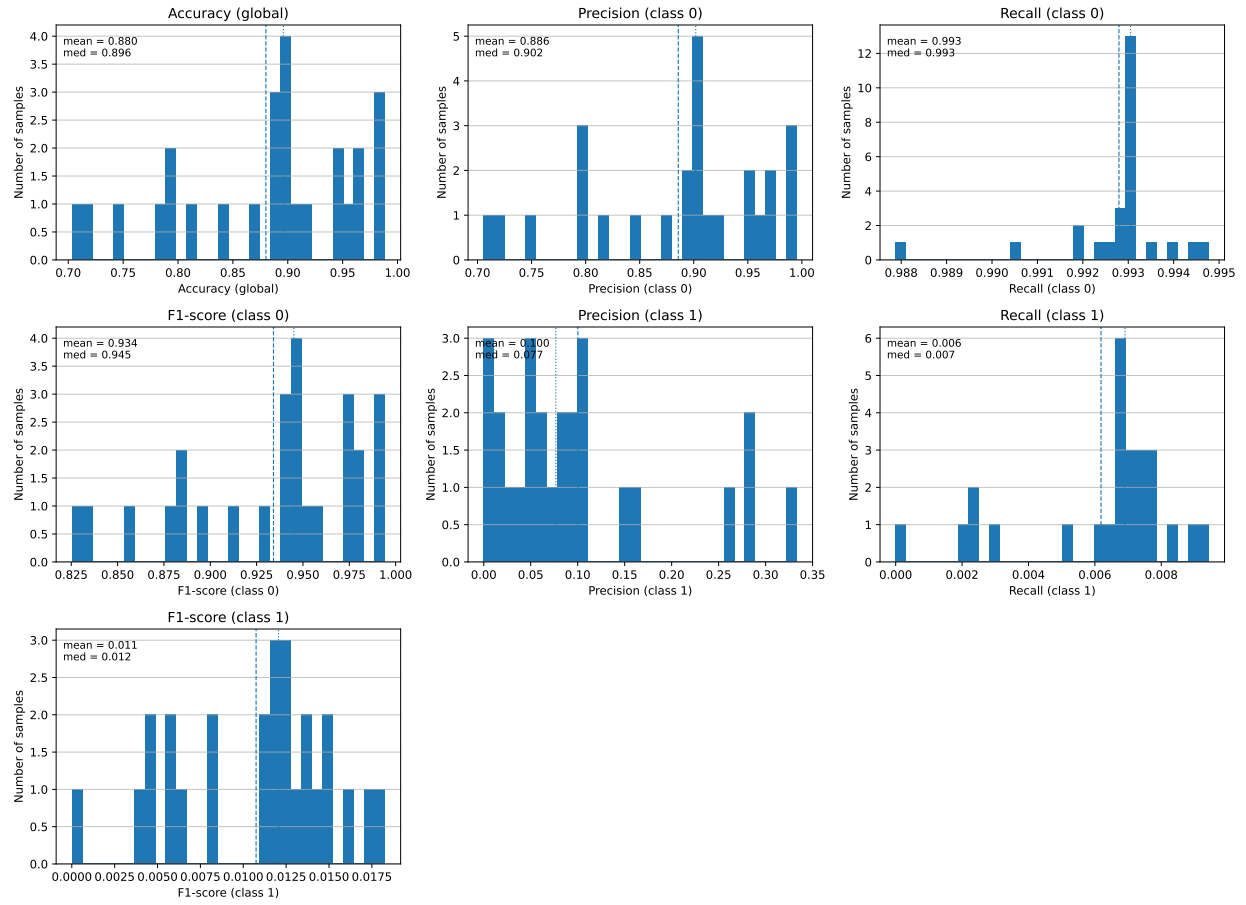


Figure 11.12: Histogram of the evaluation metrics of the random model following the baseline model's distribution on the test data

Table 11.9: Table summarizing all metrics for each train and test instance computed with the random model using a uniform distribution

File	Acc. global	Prec. 0	Recall 0	F1 0	Support 0	Prec. 1	Recall 1	F1 1	Support 1
AllInterval-006	0.4991	0.9069	0.4990	0.6438	49605	0.0925	0.4992	0.1561	5076
AllInterval-009	0.5000	0.9394	0.5000	0.6526	69539083	0.0607	0.5003	0.1083	4491537
Allergy	0.6875	0.7500	0.7500	0.7500	20	0.5833	0.5833	0.5833	12
Alpha	0.5000	0.9871	0.5000	0.6638	132012279	0.0129	0.4998	0.0251	1720383
ColouredQueens-03	0.4590	0.8750	0.4667	0.6087	165	0.0737	0.3889	0.1239	18
ColouredQueens-07	0.5000	0.9538	0.5000	0.6561	1519939348	0.0462	0.4999	0.0845	73602480
CostasArray-10	0.5000	0.9714	0.5000	0.6601	71519494	0.0286	0.5002	0.0541	2106757
Kakuro-hard-010-sumdiff	0.5046	0.9638	0.5039	0.6618	2804	0.0400	0.5225	0.0744	111
Kakuro-hard-056-sumdiff	0.5100	0.9600	0.5107	0.6667	6160	0.0404	0.4922	0.0747	258
Kakuro-hard-062-sumdiff	0.5015	0.9663	0.5000	0.6590	10272	0.0396	0.5422	0.0739	391
Kakuro-hard-137-sumdiff	0.5000	0.9681	0.5000	0.6595	146501127	0.0319	0.5004	0.0600	4827578
Kakuro-medium-035-sumdiff	0.5023	0.9691	0.5014	0.6609	44343	0.0345	0.5267	0.0648	1500
Kakuro-medium-062-sumdiff	0.4921	0.9595	0.4917	0.6502	7806	0.0397	0.5031	0.0736	326
Kakuro-medium-086-sumdiff	0.4967	0.9669	0.4950	0.6548	20684	0.0385	0.5436	0.0719	769
Kakuro-medium-129-sumdiff	0.5000	0.9640	0.5000	0.6585	43473903	0.0361	0.5006	0.0673	1624282
KnightTour-06-ext03	0.5000	0.9682	0.5000	0.6595	124103987	0.0318	0.5001	0.0598	4073809
Langford-2-06	0.5039	0.8157	0.5039	0.6229	4402	0.1890	0.5040	0.2749	1010
Langford-4-09	0.5025	0.7527	0.5017	0.6021	34024	0.2522	0.5049	0.3364	11328
MagicSquare-03-sum	0.4948	0.9238	0.4925	0.6425	5068	0.0804	0.5220	0.1394	431
OpenStacks-m1-wbop-30-10-1	0.5002	0.9574	0.5002	0.6571	7579019	0.0427	0.5001	0.0787	337802
Ortholatin-005	0.5004	0.9532	0.5004	0.6563	3120613	0.0470	0.5005	0.0859	153601
Purdey	0.4824	0.8974	0.4667	0.6140	75	0.1304	0.6000	0.2143	10
Queens-0009-m1	0.5005	0.9396	0.5007	0.6532	548012	0.0602	0.4986	0.1075	35186
Queens-0012-m1	0.5000	0.9514	0.5000	0.6555	111655796	0.0486	0.5000	0.0886	5702335
SchurrLemma-012-9-mod	0.5000	0.9460	0.5000	0.6542	55031201	0.0541	0.5003	0.0976	3144267
SchurrLemma-015-9-mod	0.4999	0.9503	0.4999	0.6552	85749044	0.0497	0.4998	0.0904	4485654

Continued on next page

Table 11.9: Table summarizing all metrics for each train and test instance computed with the random model using a uniform distribution (continued)

File	Acc. global	Prec. 0	Recall 0	F1 0	Support 0	Prec. 1	Recall 1	F1 1	Support 1
SchurrLemma-020-9-mod	0.5000	0.9599	0.5000	0.6575	79308261	0.0400	0.4995	0.0741	3309219
SchurrLemma-030-9-mod	0.5000	0.9685	0.5000	0.6595	187707401	0.0314	0.4997	0.0592	6097149
SchurrLemma-050-9-mod	0.5000	0.9788	0.5000	0.6619	521797731	0.0212	0.5000	0.0407	11310129
SchurrLemma-100-9-mod	0.5000	0.9882	0.5000	0.6640	2041227181	0.0118	0.4999	0.0230	24342579
Subisomorphism-g16-g46	0.5002	0.9939	0.5002	0.6655	14358706	0.0061	0.5013	0.0121	87899
Subisomorphism-si2-m4Dr6-m625-04	0.4999	0.9954	0.4999	0.6656	17591559	0.0046	0.4979	0.0090	80909
Subisomorphism-si4-m4Dr2-m625-02	0.4995	0.9875	0.4995	0.6634	1252654	0.0124	0.4985	0.0243	15834
Sudoku-s03b-alldiff	0.5161	0.7998	0.5215	0.6313	1701	0.2112	0.4955	0.2962	440
Sudoku-s08c-alldiff	0.5077	0.8073	0.5022	0.6192	1852	0.2133	0.5297	0.3041	472
Sudoku-s11b-alldiff	0.5063	0.8473	0.5086	0.6357	2432	0.1537	0.4932	0.2343	440
Sudoku-s11c-alldiff	0.4979	0.8006	0.4943	0.6113	1681	0.2034	0.5118	0.2911	424
Sudoku-s12a-alldiff	0.4872	0.8666	0.4899	0.6260	3329	0.1152	0.4682	0.1849	472
Sudoku-s13a-alldiff	0.5004	0.7033	0.5048	0.5877	939	0.2922	0.4898	0.3661	392
Sudoku-s13c-alldiff	0.5118	0.7284	0.5164	0.6044	914	0.2848	0.5000	0.3629	352
bqwh-15-106-02_X2	0.5018	0.9038	0.4998	0.6437	25742	0.1032	0.5196	0.1722	2852
bqwh-15-106-07_X2	0.4993	0.9035	0.4988	0.6427	89078	0.0974	0.5037	0.1632	9564
bqwh-15-106-25_X2	0.4993	0.9040	0.4993	0.6433	216852	0.0955	0.4992	0.1604	22971
bqwh-15-106-51_X2	0.5006	0.8979	0.5009	0.6431	87038	0.1015	0.4975	0.1686	9866
bqwh-15-106-81_X2	0.5021	0.8997	0.5008	0.6435	45132	0.1055	0.5130	0.1750	5177
bqwh-18-141-50_X2	0.4995	0.9101	0.4992	0.6448	217515	0.0904	0.5024	0.1532	21547
bqwh-18-141-75_X2	0.5028	0.9097	0.5023	0.6473	265270	0.0935	0.5074	0.1580	26850
sudoku368_X2	0.5012	0.9720	0.5006	0.6608	52489	0.0306	0.5218	0.0578	1585

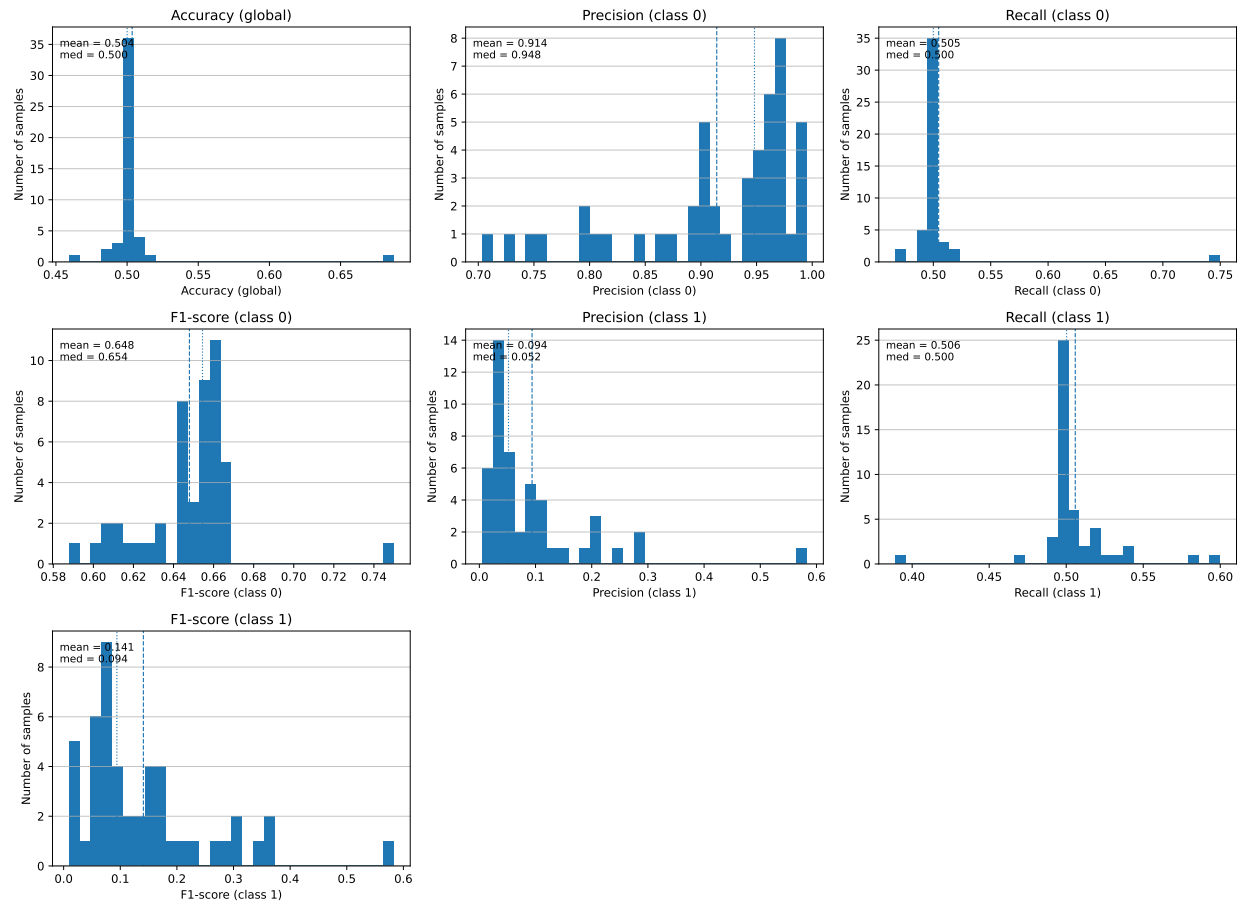


Figure 11.13: Histogram of the evaluation metrics of the random model with a uniform distribution on the training and test data

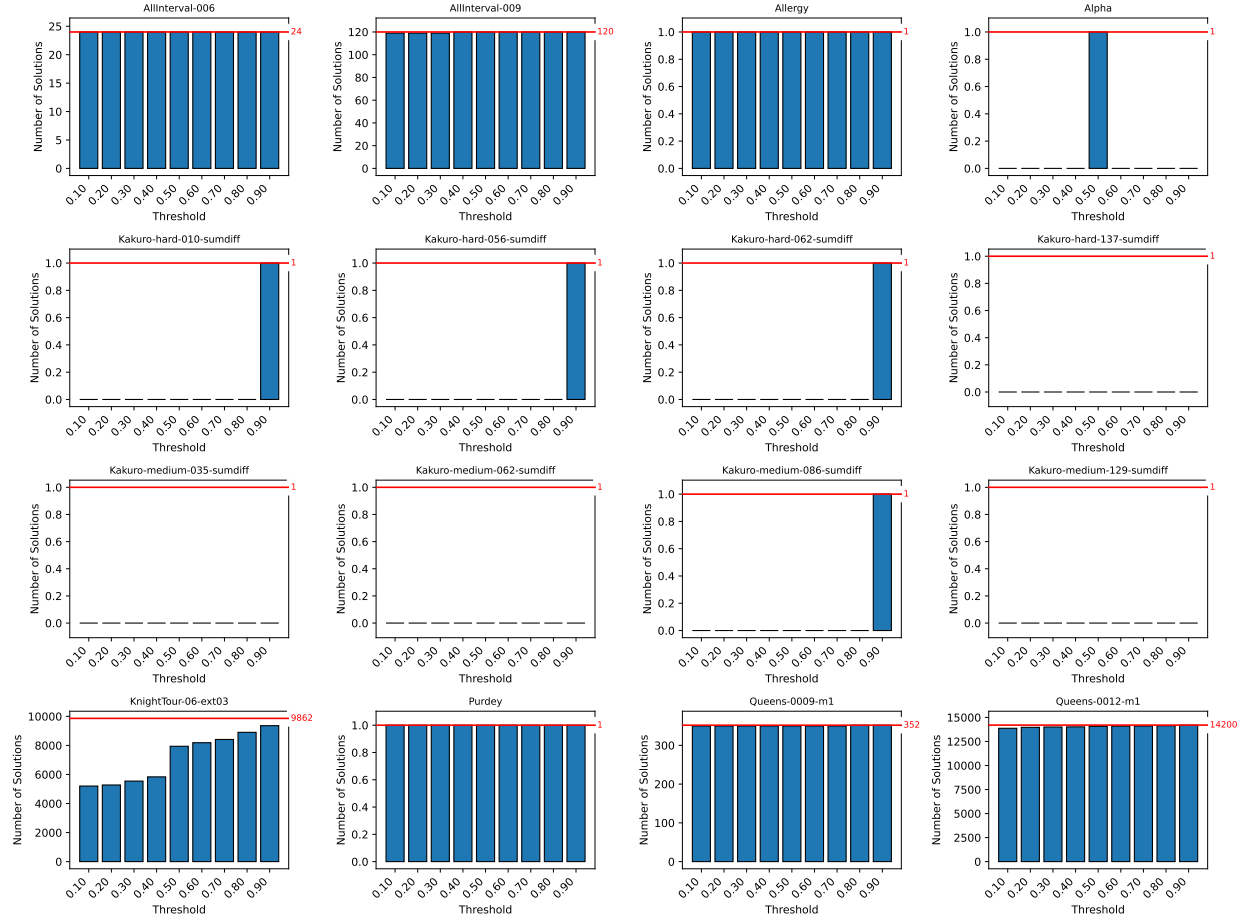


Figure 11.14: Number of solutions as a function of the threshold for the baseline model, evaluated on the training instances

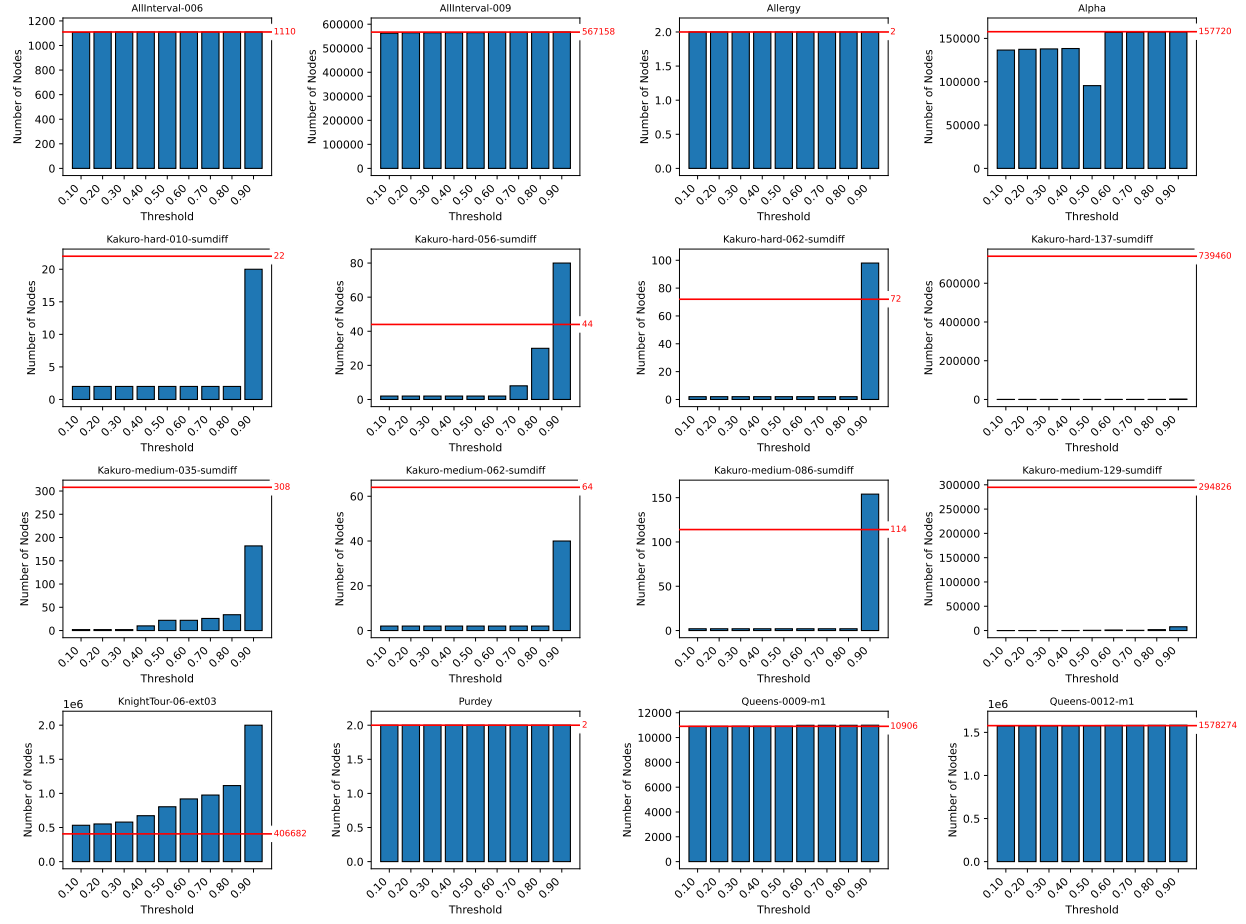


Figure 11.15: Number of nodes as a function of the threshold for the baseline model, evaluated on the training instances

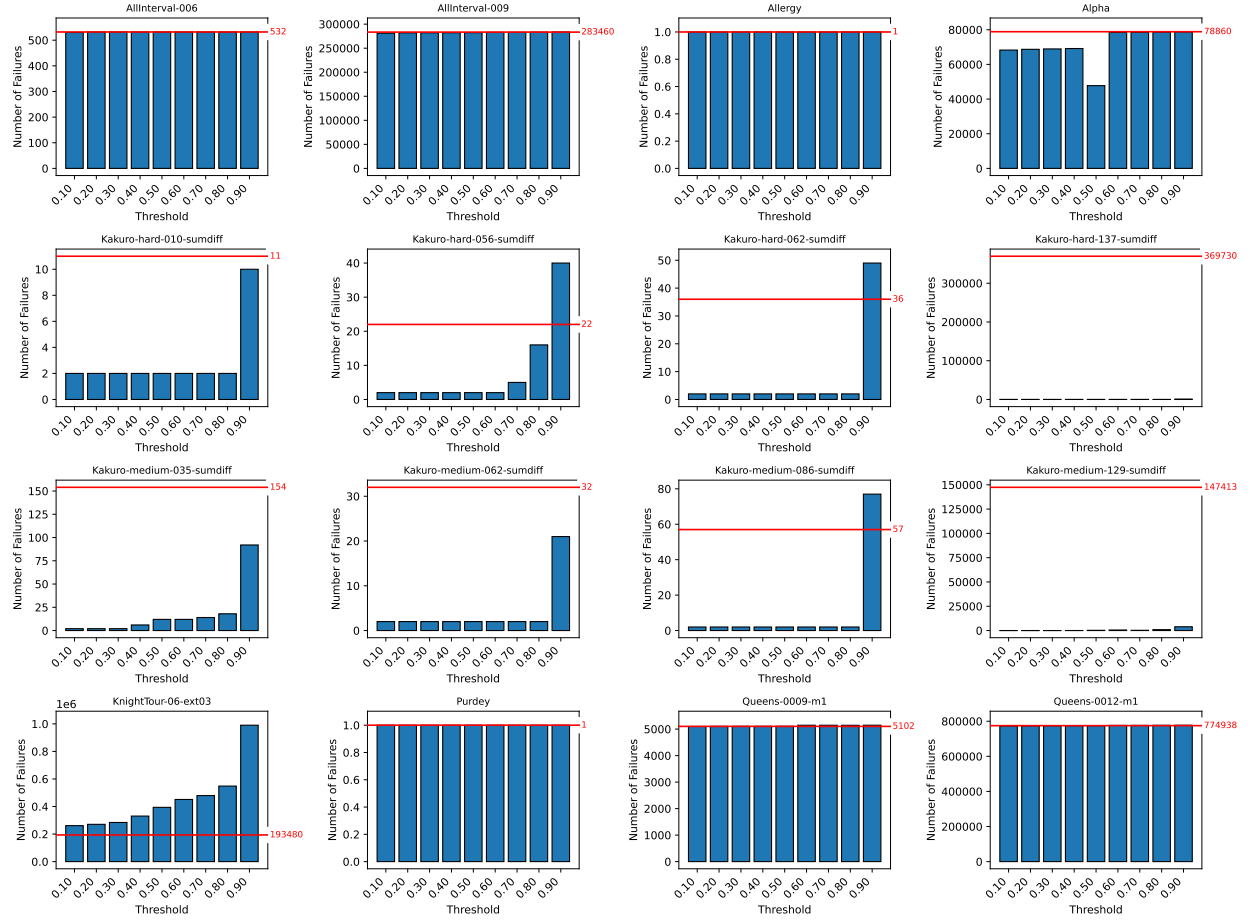


Figure 11.16: Number of failures as a function of the threshold for the baseline model, evaluated on the training instances

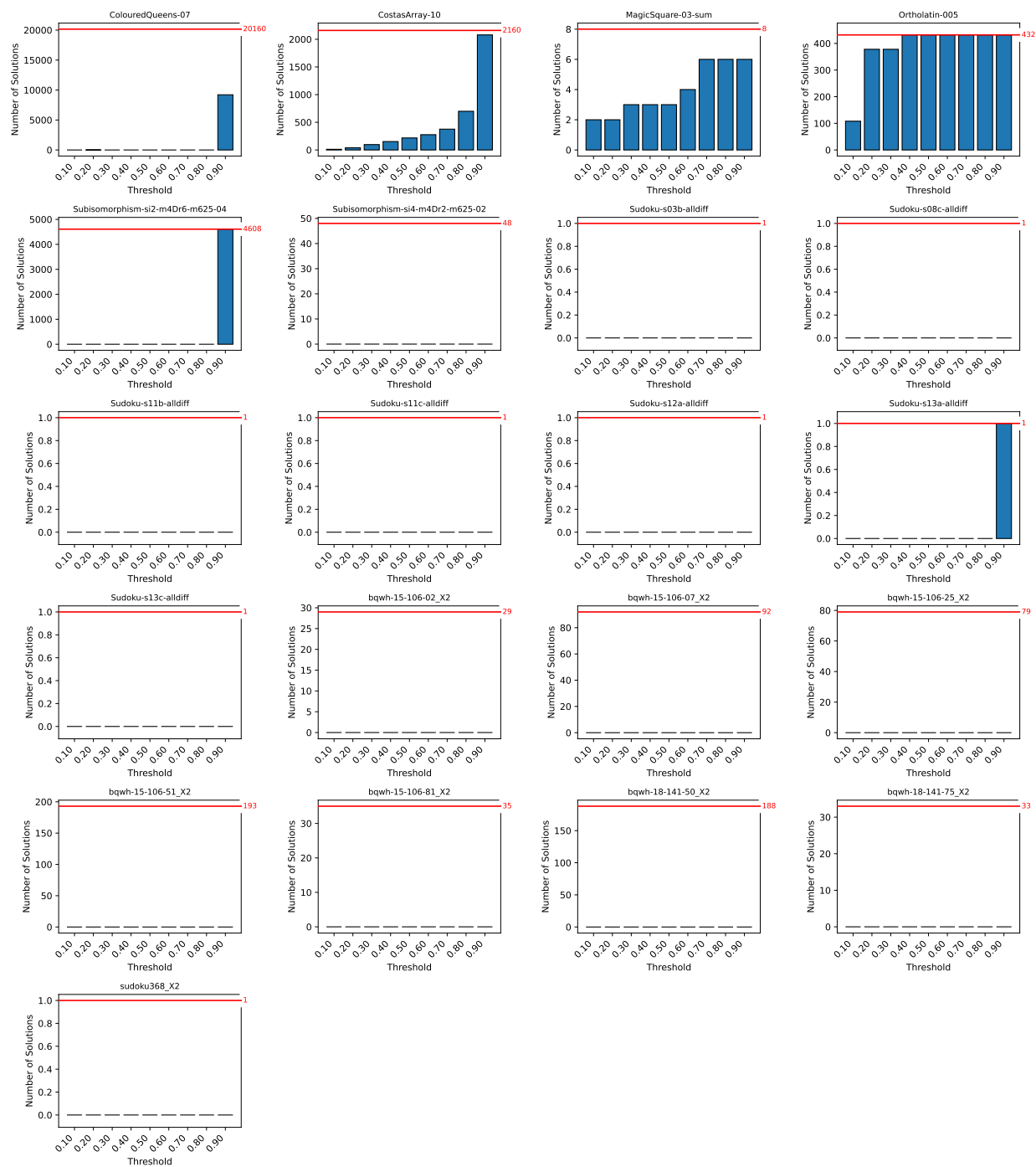


Figure 11.17: Number of solutions as a function of the threshold for the baseline model, evaluated on the test instances

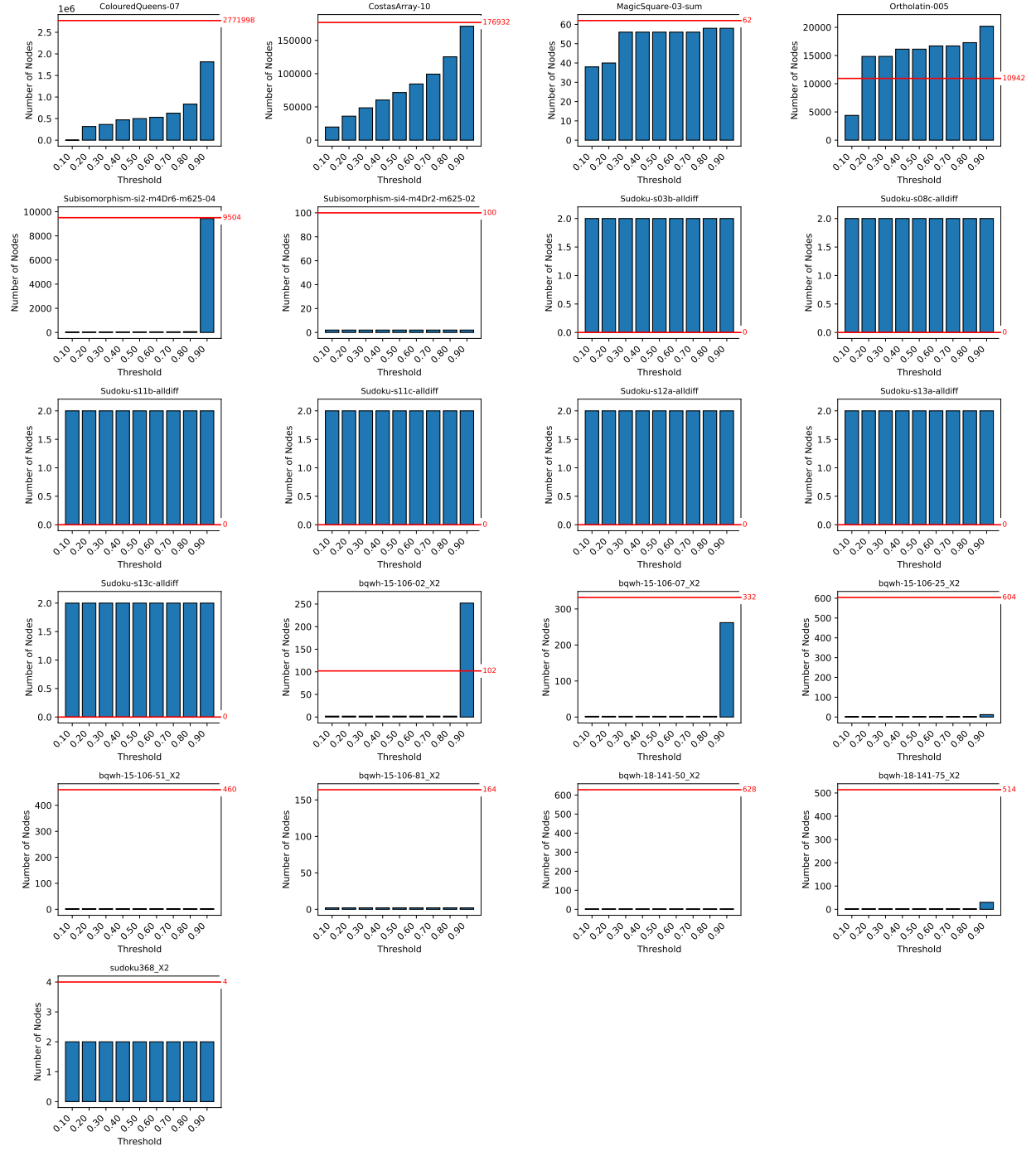


Figure 11.18: Number of nodes as a function of the threshold for the baseline model, evaluated on the test instances

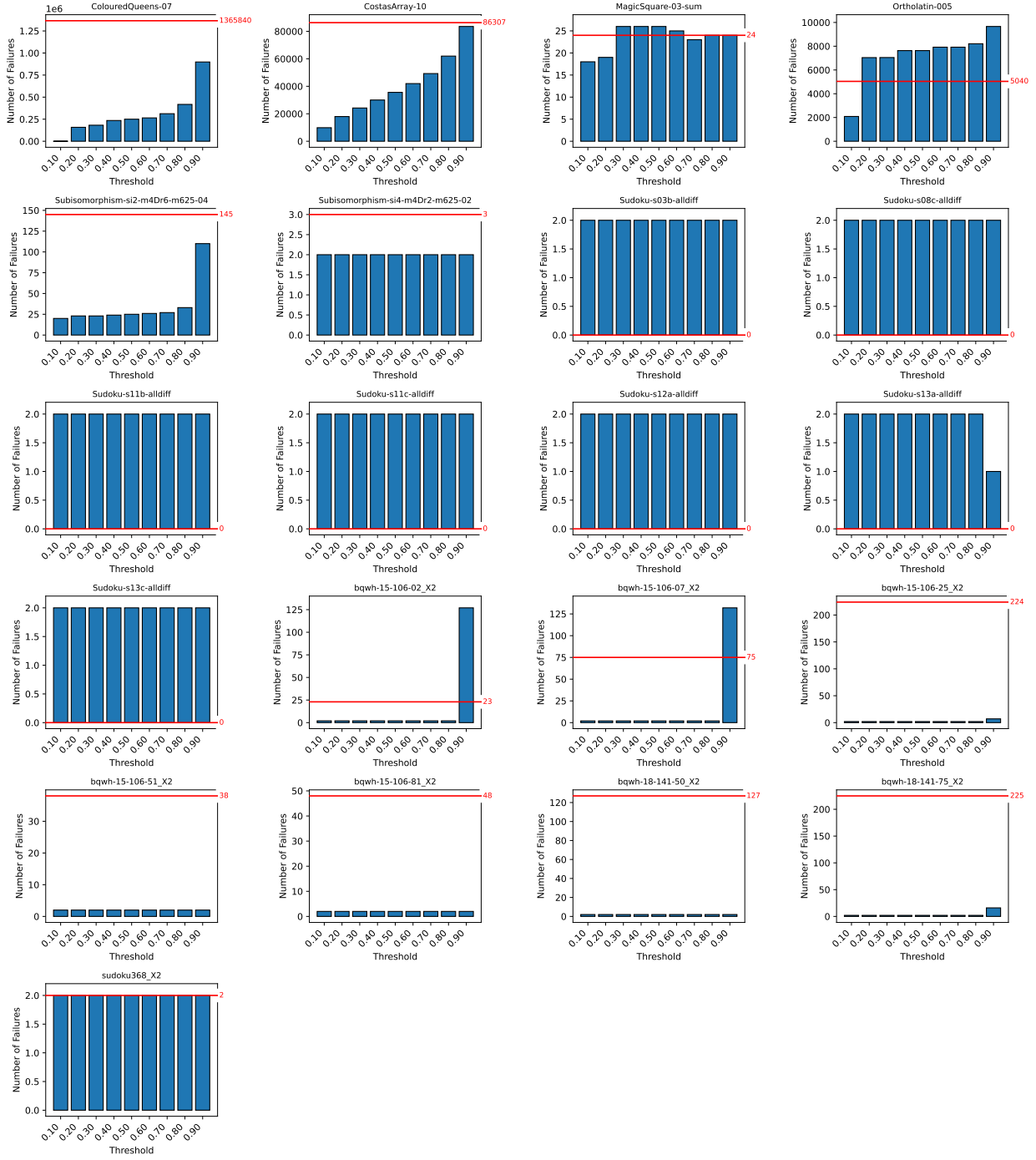


Figure 11.19: Number of failures as a function of the threshold for the baseline model, evaluated on the test instances

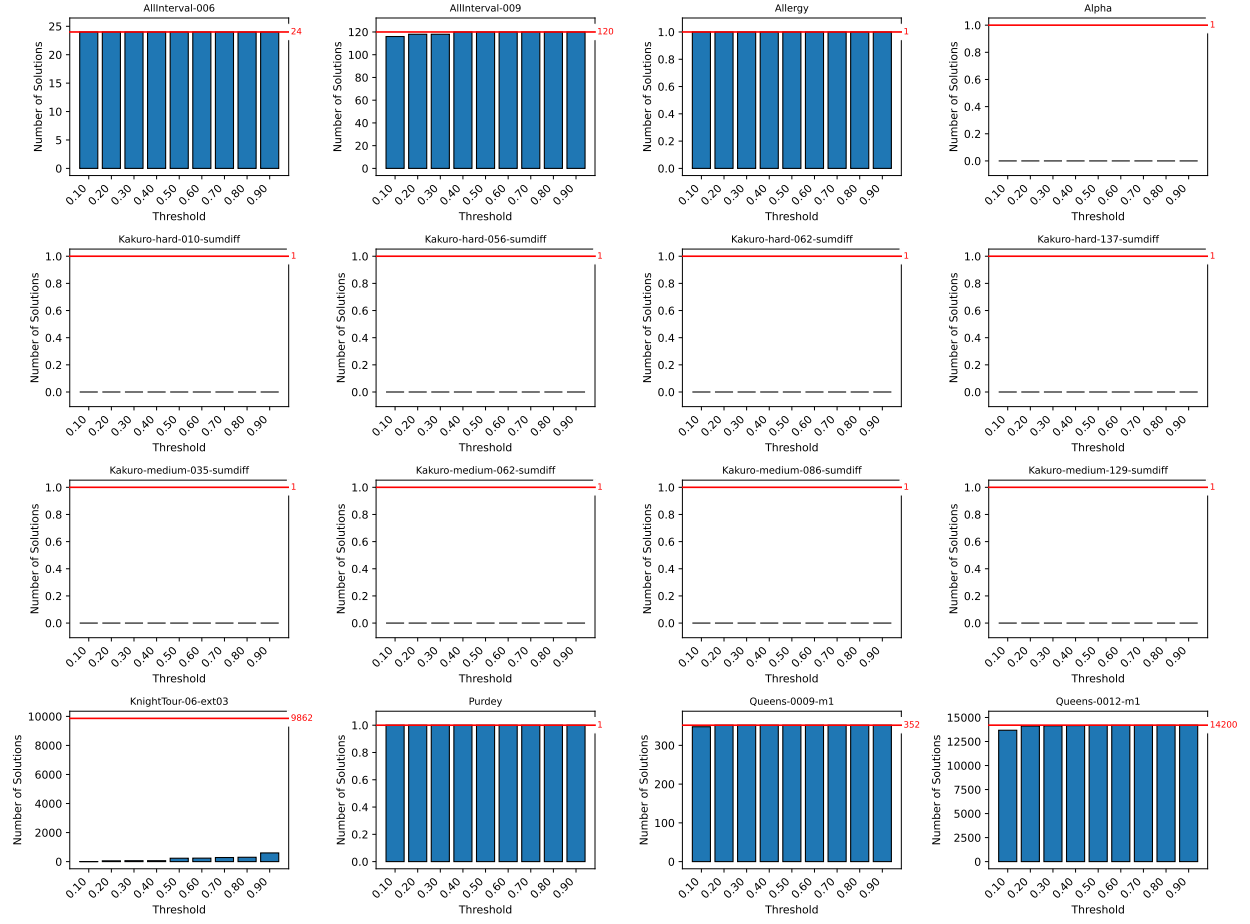


Figure 11.20: Number of solutions as a function of the threshold for the small model, evaluated on the training instances

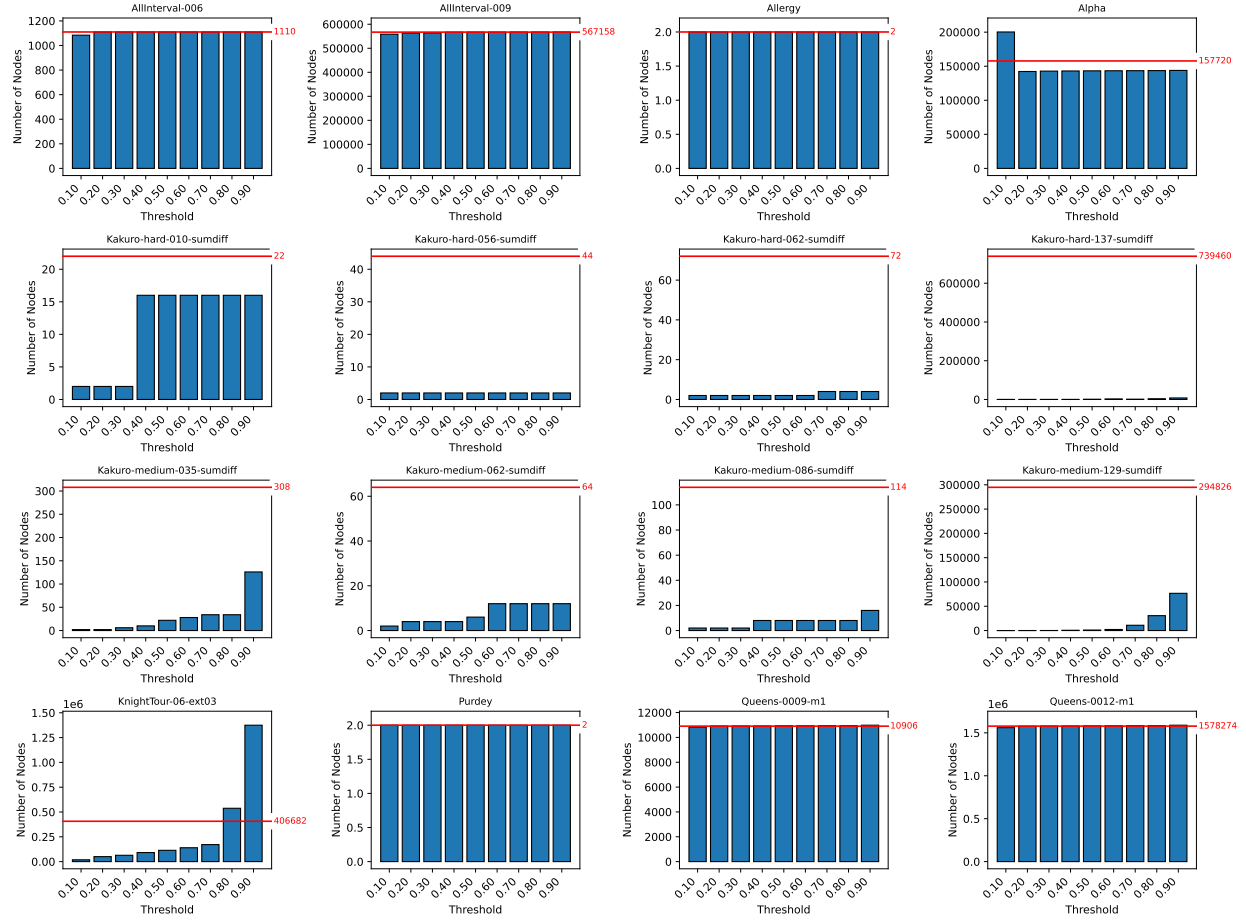


Figure 11.21: Number of nodes as a function of the threshold for the small model, evaluated on the training instances

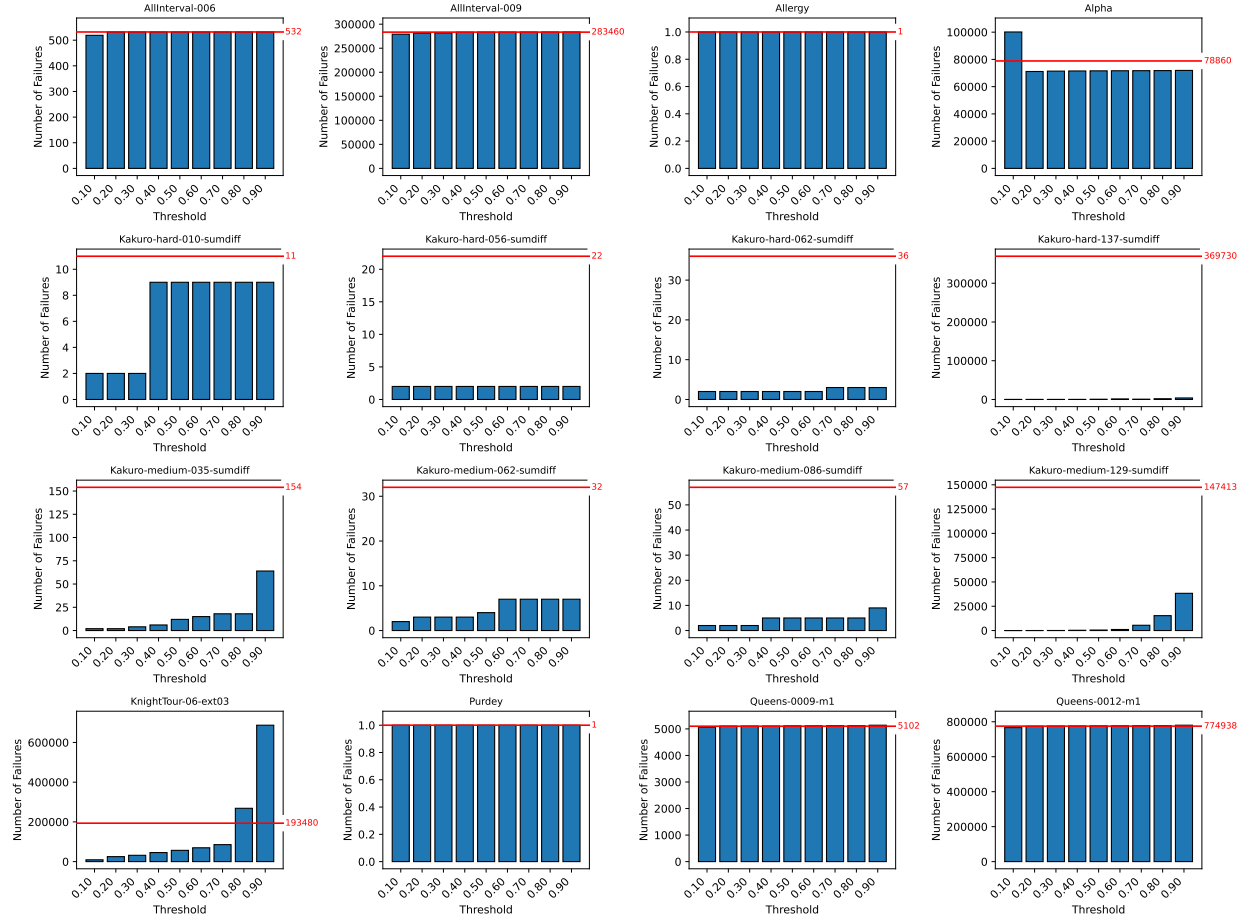


Figure 11.22: Number of failures as a function of the threshold for the small model, evaluated on the training instances

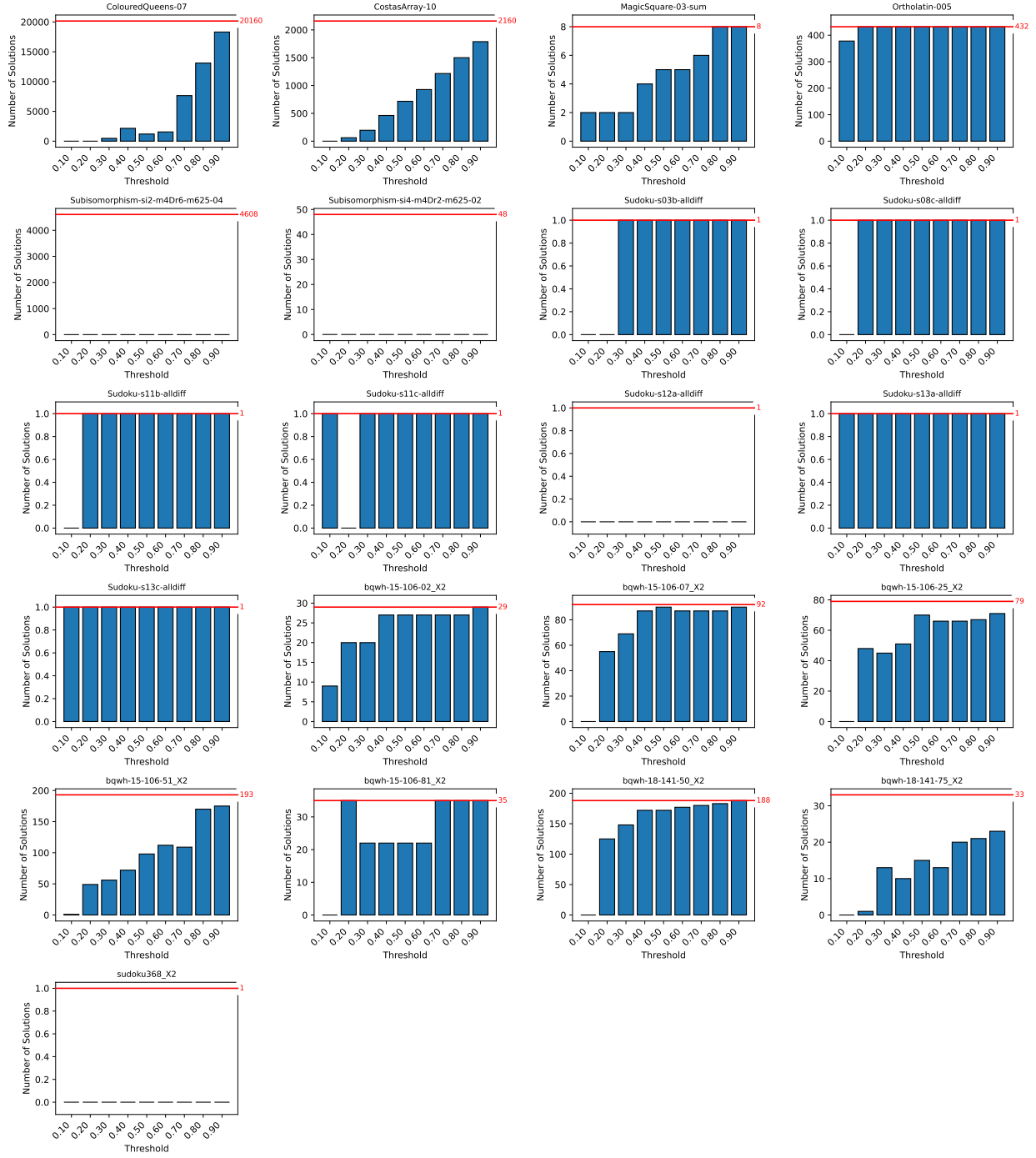


Figure 11.23: Number of solutions as a function of the threshold for the small model, evaluated on the test instances

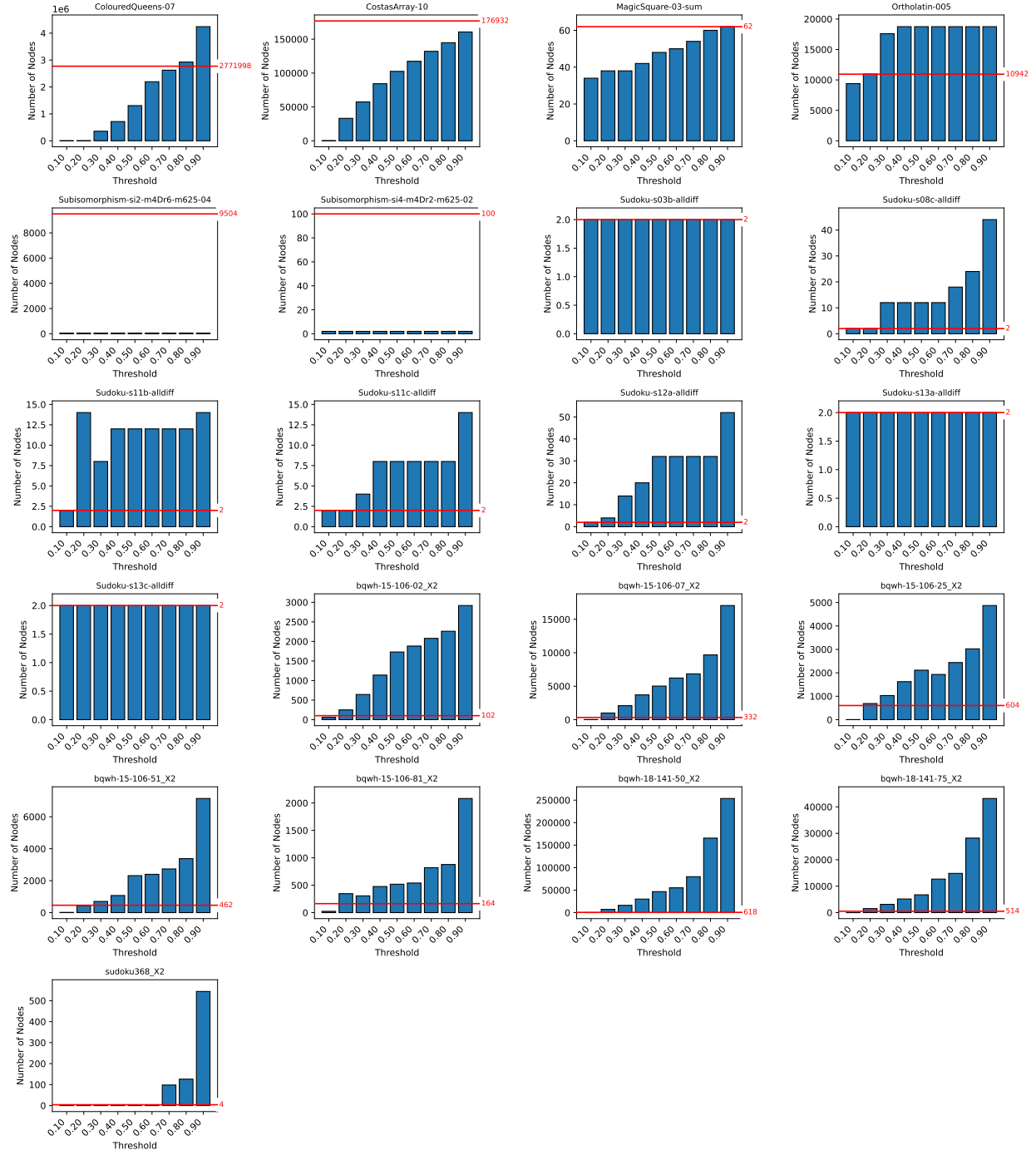


Figure 11.24: Number of nodes as a function of the threshold for the small model, evaluated on the test instances

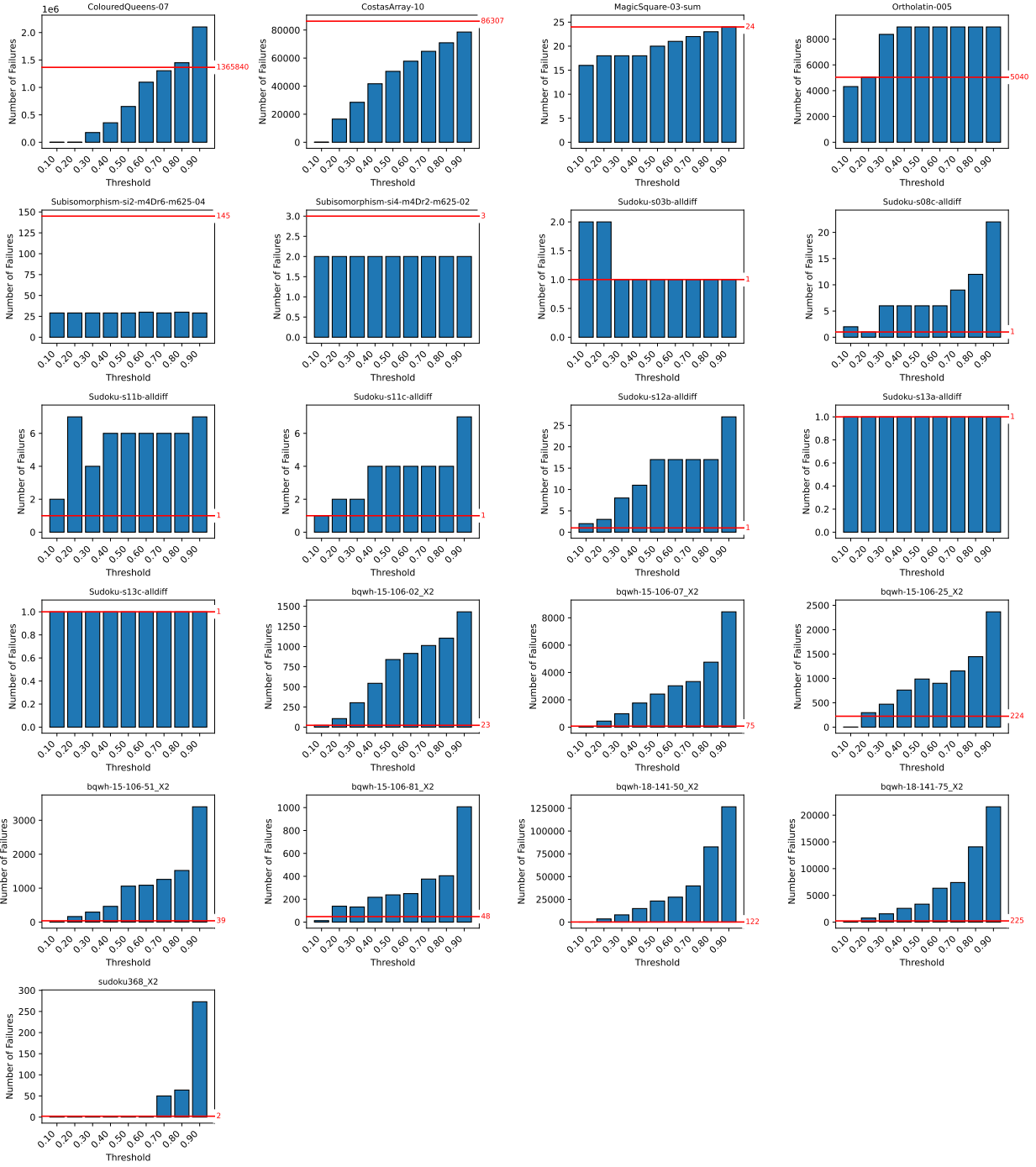


Figure 11.25: Number of failures as a function of the threshold for the small model, evaluated on the test instances

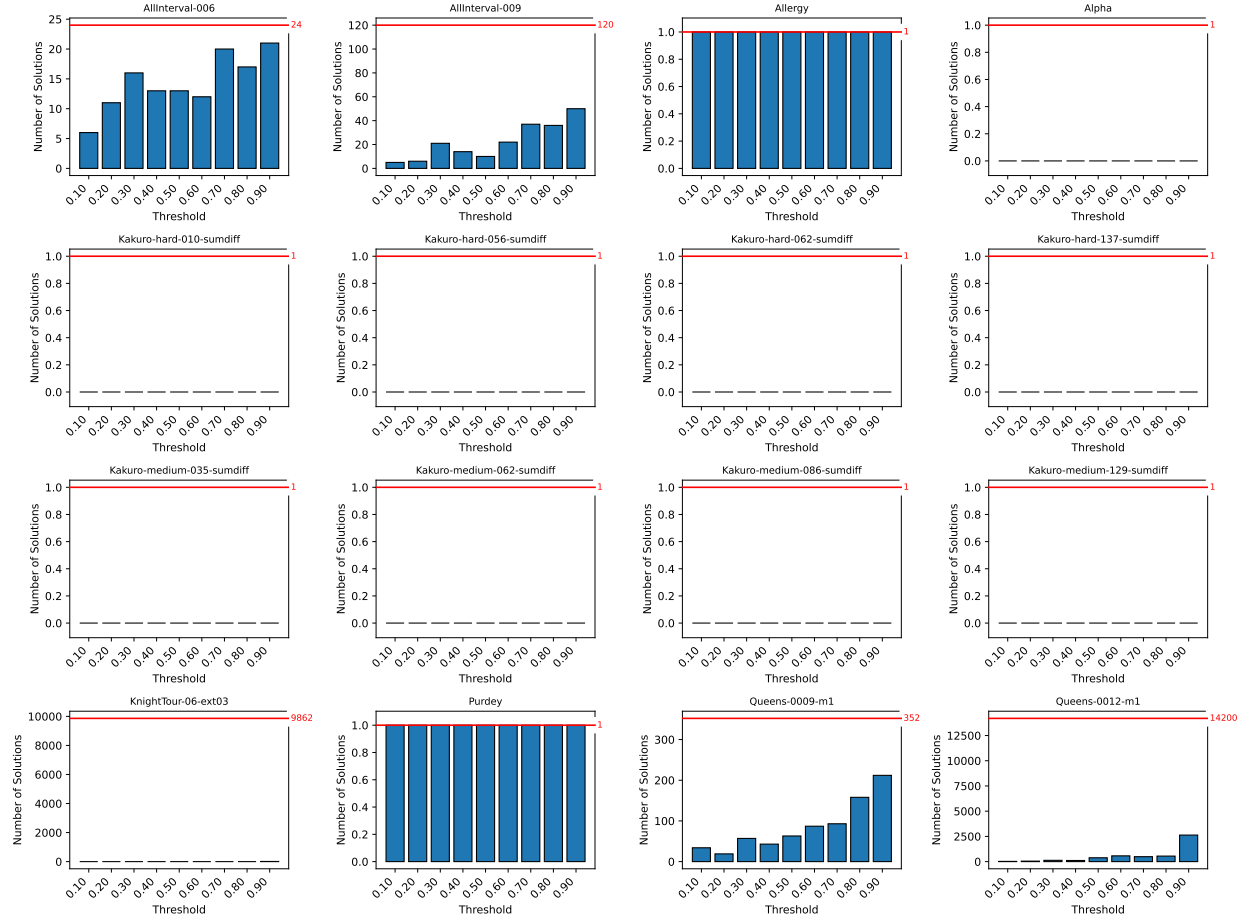


Figure 11.26: Number of solutions as a function of the threshold for the random model following the baseline model's distribution, evaluated on the training instances

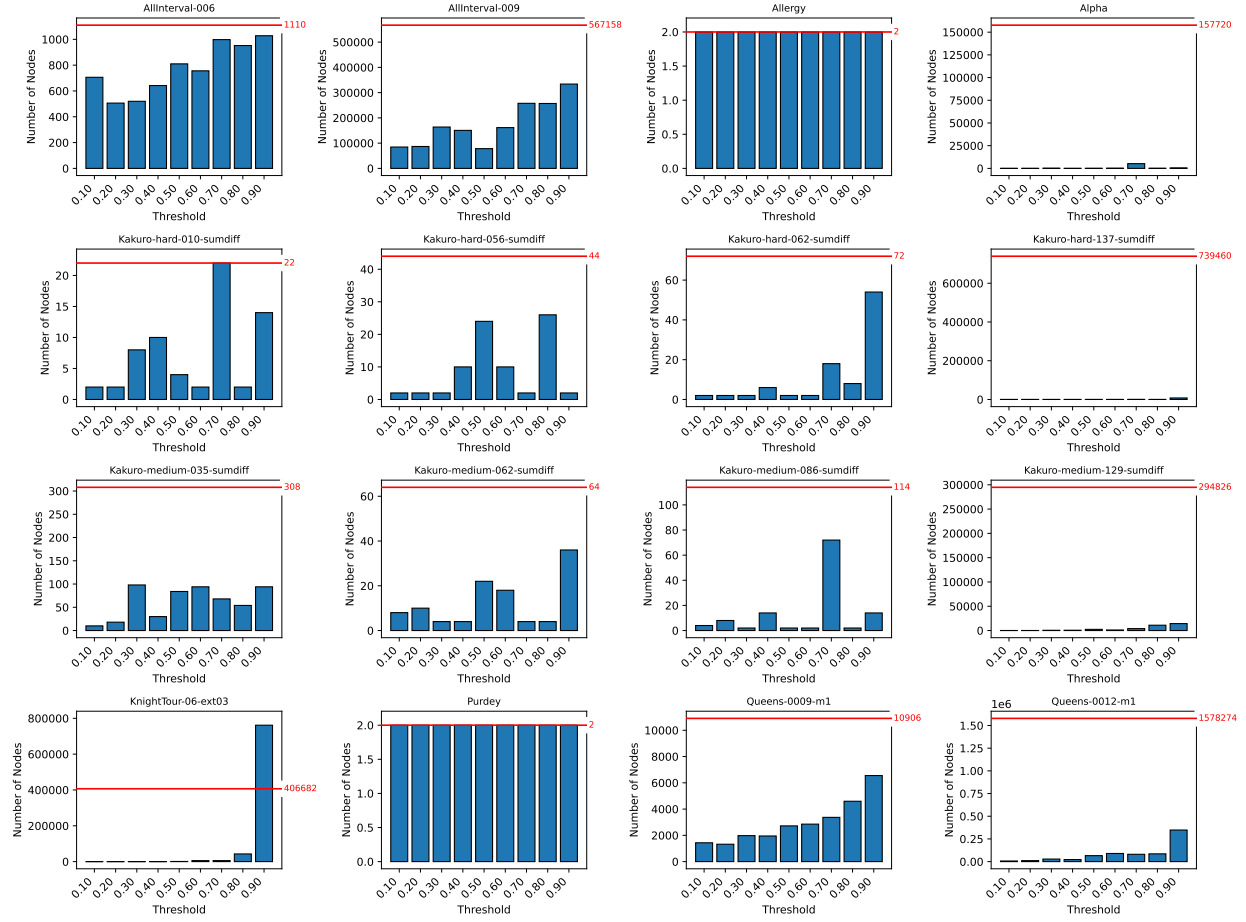


Figure 11.27: Number of nodes as a function of the threshold for the random model following the baseline model's distribution, evaluated on the training instances

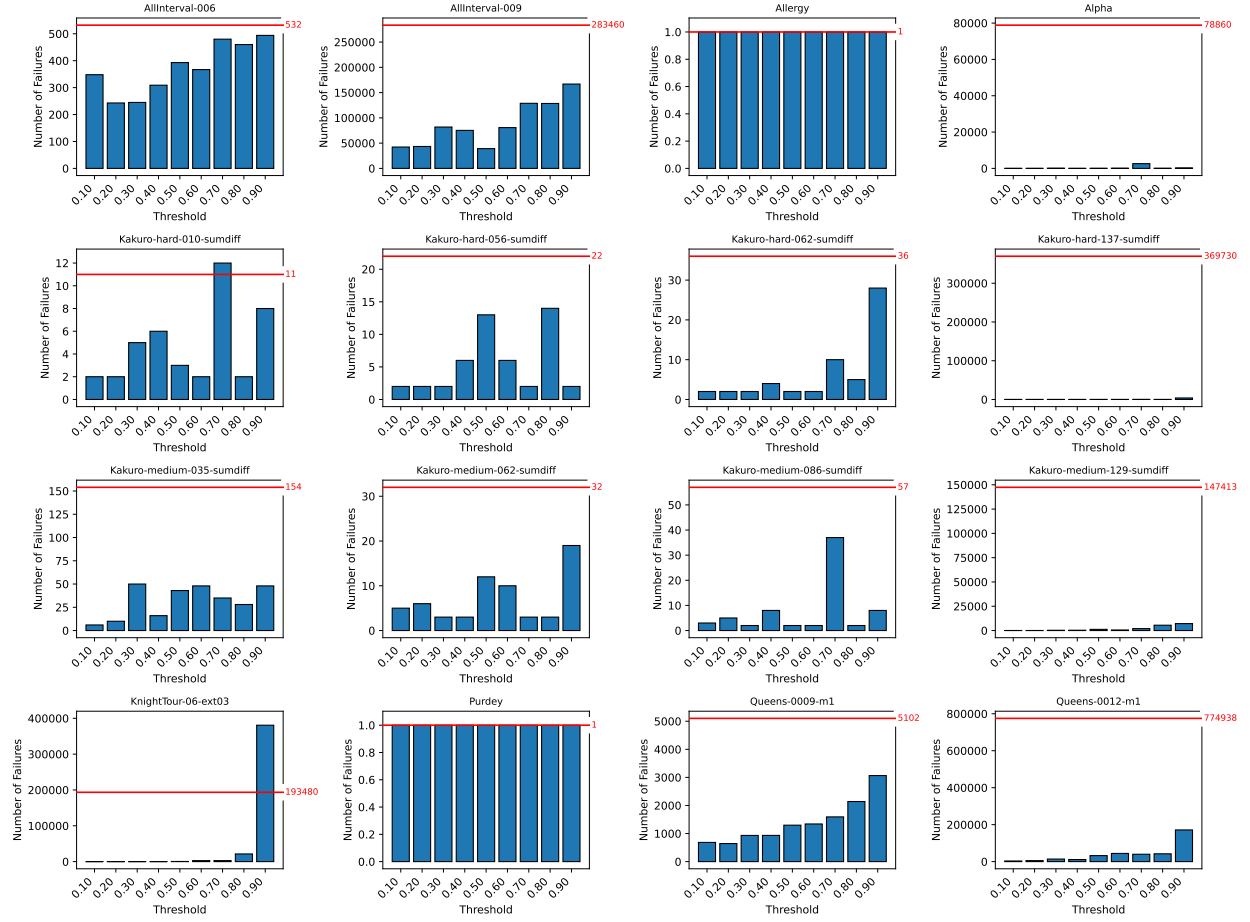


Figure 11.28: Number of failures as a function of the threshold for the random model following the baseline model's distribution, evaluated on the training instances

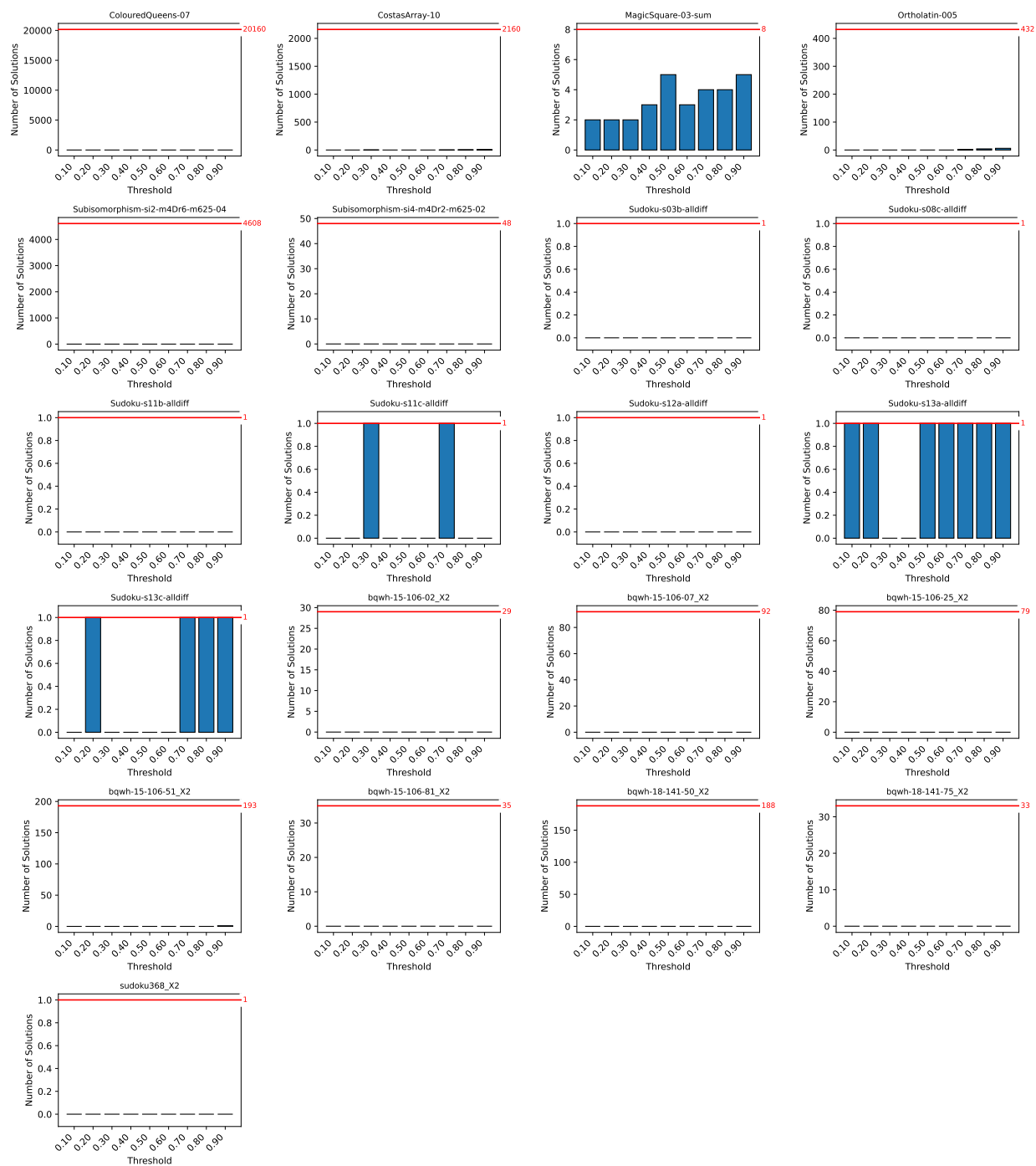


Figure 11.29: Number of solutions as a function of the threshold for the random model following the baseline model's distribution, evaluated on the test instances

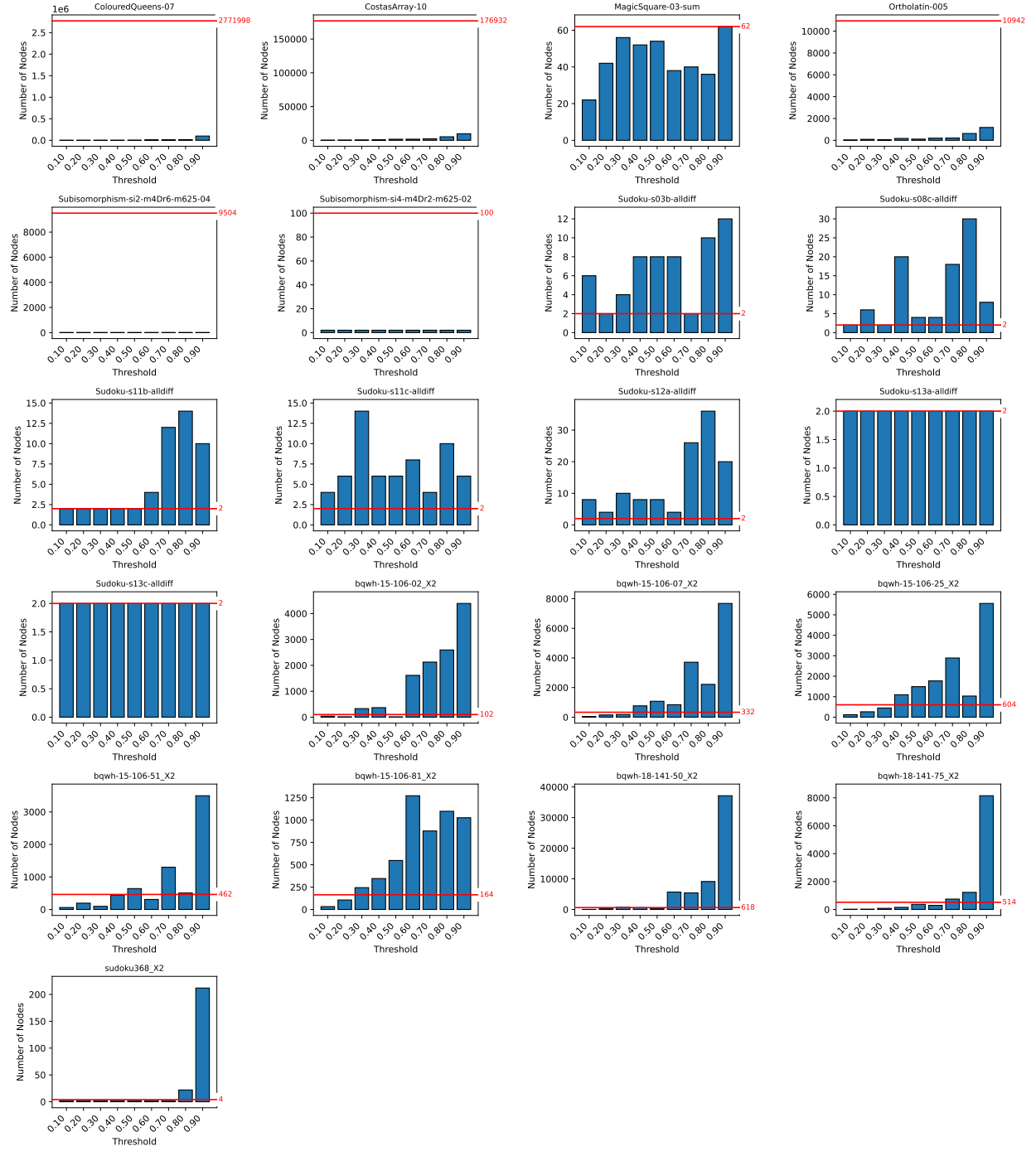


Figure 11.30: Number of nodes as a function of the threshold for the random model following the baseline model's distribution, evaluated on the test instances

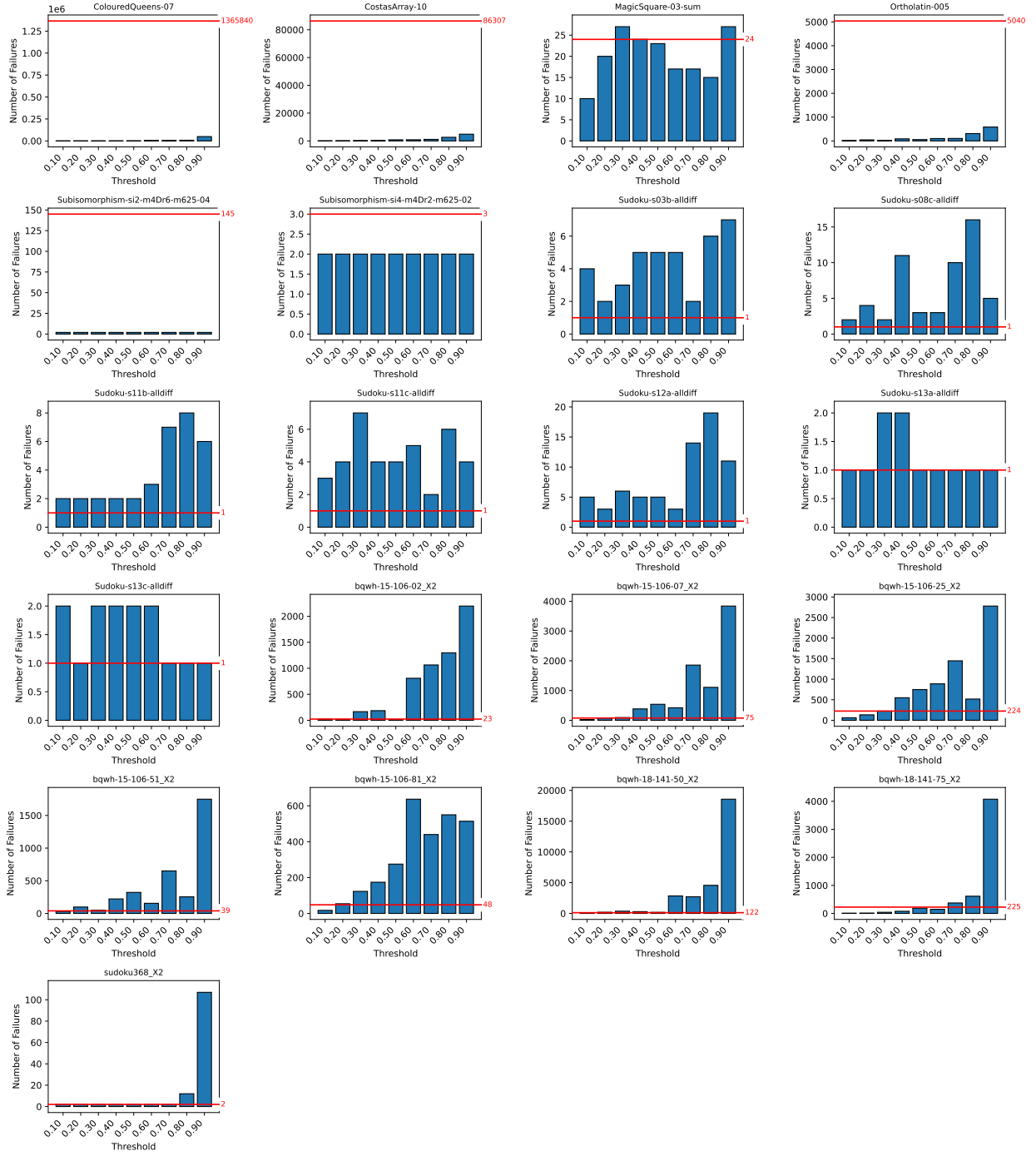


Figure 11.31: Number of failures as a function of the threshold for the random model following the baseline model's distribution, evaluated on the test instances

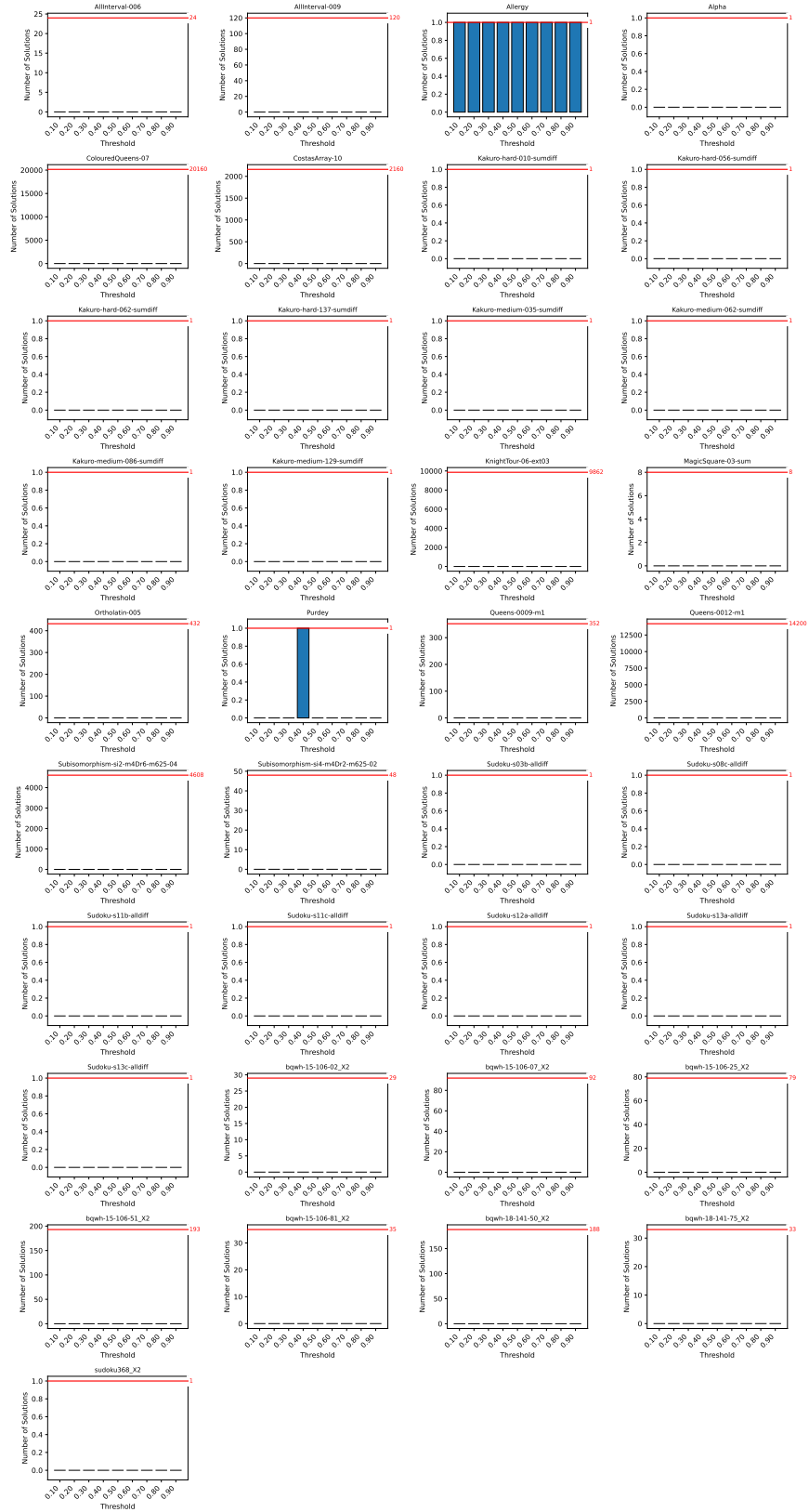


Figure 11.32: Number of solutions as a function of the threshold for the random model with a uniform distribution, evaluated on the training and test instances

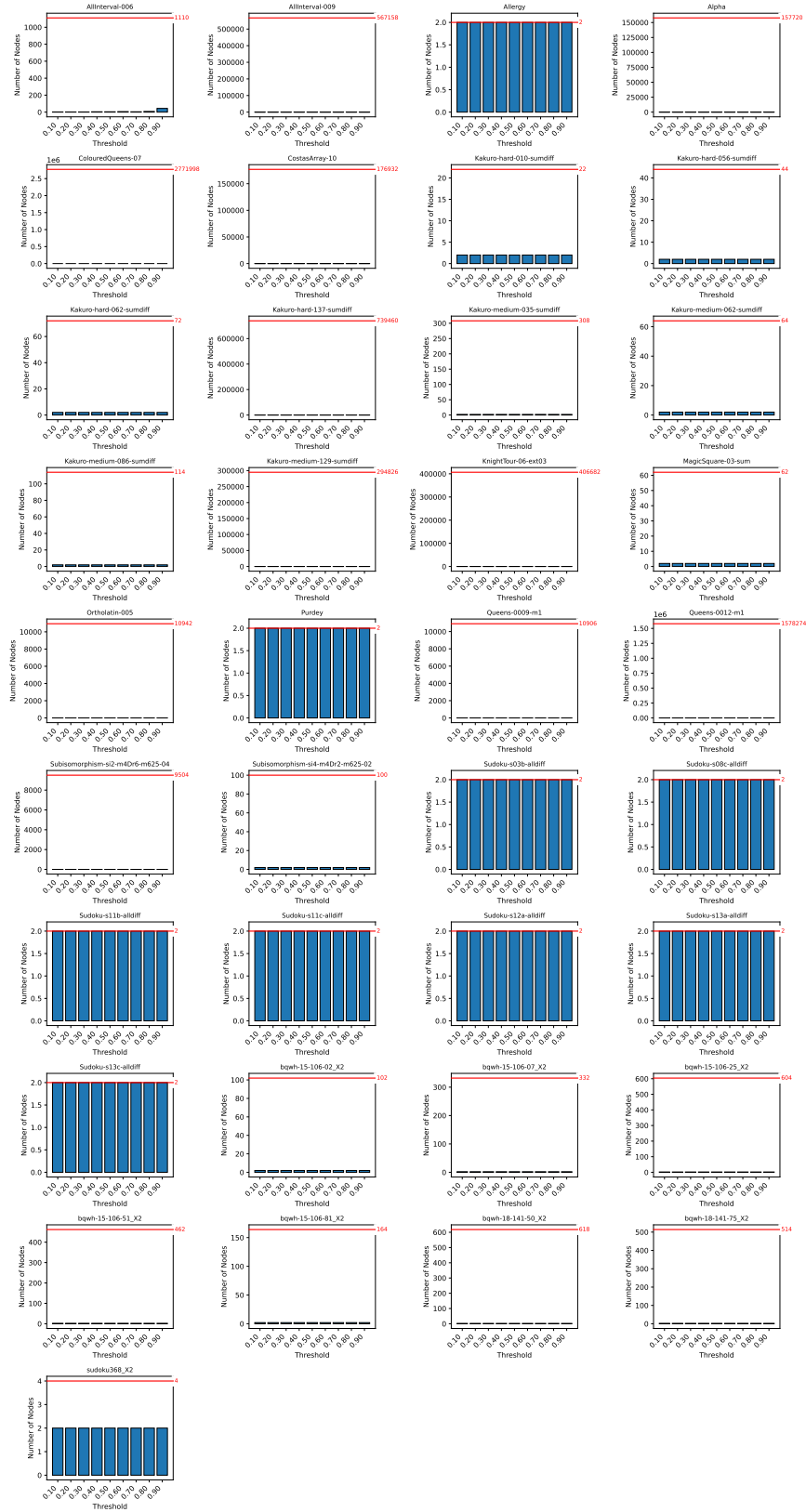


Figure 11.33: Number of nodes as a function of the threshold for the random model with a uniform distribution, evaluated on the training and test instances

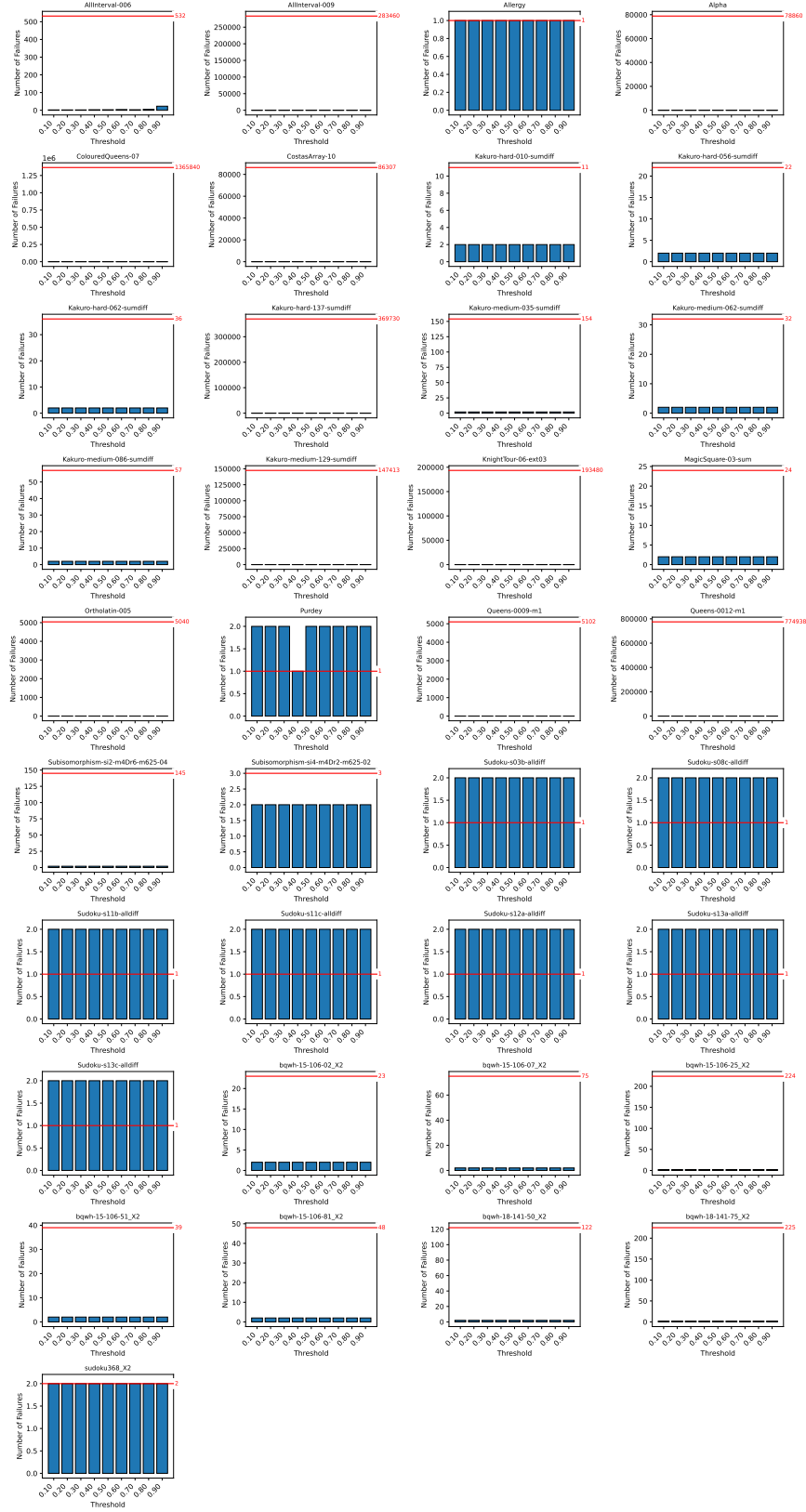


Figure 11.34: Number of failures as a function of the threshold for the random model with a uniform distribution, evaluated on the training and test instances

Table 11.10: Comparison of execution time between the baseline solver without machine learning and the solver using the baseline model with a threshold of 0.5 on the training instances (A: Total AI / Solver without AI total, B: Python AI / # propagation (ms))

Instance	Solver without AI total (ms)	AI total (ms)	AI Java (ms)	AI Python (ms)	A	B
AllInterval-006	7	1367.00	17.00	1223.00	195.29	0.452293
AllInterval-009	4612	829481.00	10723.00	739279.00	179.85	0.512152
Allergy	4	0.00	0.00	0.00	0.00	0.000000
Alpha	3279	209935.00	3140.00	181312.00	64.02	0.837449
Kakuro-hard-010-sumdiff	16	40.00	3.00	31.00	2.50	0.968750
Kakuro-hard-056-sumdiff	6	23.00	3.00	17.00	3.83	0.425000
Kakuro-hard-062-sumdiff	11	8.00	2.00	5.00	0.73	0.416667
Kakuro-hard-137-sumdiff	13522	26.00	1.00	23.00	0.00	0.522727
Kakuro-medium-035-sumdiff	11	149.00	2.00	134.00	13.55	0.492647
Kakuro-medium-062-sumdiff	7	20.00	2.00	14.00	2.86	0.451613
Kakuro-medium-086-sumdiff	10	7.00	1.00	5.00	0.70	0.357143
Kakuro-medium-129-sumdiff	5349	4384.00	44.00	3934.00	0.82	0.450992
KnightTour-06-ext03	10965	1597954.00	34306.00	1392334.00	145.73	0.557166
Purdey	22	19.00	0.00	17.00	0.86	2.125000
Queens-0009-m1	193	15883.00	347.00	14075.00	82.30	0.472807
Queens-0012-m1	33658	2376713.00	57744.00	2095195.00	70.61	0.474478

Table 11.11: Comparison of execution time between the baseline solver without machine learning and the solver using the baseline model with a threshold of 0.5 on the test instances (A: Total AI / Solver without AI total, B: Python AI / # propagation (ms))

Instance	Solver without AI			AI total (ms)	AI Java (ms)	AI Python (ms)	A	B
ColouredQueens-07	102674	102378	47.00	52433.00	9211347.00	99.71	0.486618	
CostasArray-10	6279	589707.00		5342.00	527672.00	93.92	0.531121	
MagicSquare-03-sum	8	71.00	1.00		62.00	8.88	0.476923	
Ortholatin-005	324	104006.00	1125.00		92844.00	321.01	0.414721	
Subisomorphism-si2-m4Dr6-m625-04	2794	15036.00	1265.00		12561.00	5.38	67.171123	
Subisomorphism-si4-m4Dr2-m625-02	729	697.00	229.00		426.00	0.96	142.000000	
Sudoku-s03b-alldiff	16	12.00	1.00		9.00	0.75	0.450000	
Sudoku-s08c-alldiff	5	18.00	0.00		16.00	3.60	0.551724	
Sudoku-s11b-alldiff	3	19.00	0.00		16.00	6.33	0.421053	
Sudoku-s11c-alldiff	6	17.00	1.00		14.00	2.83	0.518519	
Sudoku-s12a-alldiff	3	11.00	0.00		9.00	3.67	0.473684	
Sudoku-s13a-alldiff	5	11.00	0.00		9.00	2.20	0.257143	
Sudoku-s13c-alldiff	3	5.00	0.00		4.00	1.67	0.363636	
bqwh-15-106-02_X2	8	33.00	0.00		29.00	4.12	0.500000	
bqwh-15-106-07_X2	38	21.00	0.00		18.00	0.55	0.514286	
bqwh-15-106-25_X2	74	19.00	0.00		17.00	0.26	0.531250	
bqwh-15-106-51_X2	52	13.00	0.00		11.00	0.25	0.578947	
bqwh-15-106-81_X2	10	8.00	0.00		6.00	0.80	0.545455	
bqwh-18-141-50_X2	266	27.00	2.00		14.00	0.10	0.666667	
bqwh-18-141-75_X2	106	20.00	0.00		17.00	0.19	0.459459	
sudoku368_X2	47	23.00	1.00		16.00	0.49	1.066667	

Table 11.12: Comparison of execution time between the baseline solver without machine learning and the solver using the small model with a threshold of 0.5 on the training instances (A: Total AI / Solver without AI total, B: Python AI / # propagation (ms))

Instance	Solver without AI total (ms)		AI total (ms)	AI Java (ms)	AI Python (ms)	A	B
AllInterval-006		7	1311.00	20.00	1152.00	187.29	0.425249
AllInterval-009		4612	786804.00	11444.00	690090.00	170.60	0.475626
Allergy		4	0.00	0.00	0.00	0.00	0.000000
Alpha		3279	281116.00	4567.00	239991.00	85.73	0.791681
Kakuro-hard-010-sumdiff		16	112.00	1.00	98.00	7.00	0.352518
Kakuro-hard-056-sumdiff		6	31.00	3.00	19.00	5.17	0.475000
Kakuro-hard-062-sumdiff		11	39.00	1.00	34.00	3.55	0.404762
Kakuro-hard-137-sumdiff		13522	6271.00	73.00	5529.00	0.46	0.398975
Kakuro-medium-035-sumdiff		11	132.00	1.00	116.00	12.00	0.412811
Kakuro-medium-062-sumdiff		7	92.00	1.00	80.00	13.14	0.418848
Kakuro-medium-086-sumdiff		10	66.00	1.00	57.00	6.60	0.318436
Kakuro-medium-129-sumdiff		5349	8822.00	100.00	7791.00	1.65	0.427045
KnightTour-06-ext03		10965	207406.00	4773.00	179162.00	18.92	0.501002
Purdey		22	3.00	0.00	2.00	0.14	0.250000
Queens-0009-m1		193	14739.00	346.00	12826.00	76.37	0.430042
Queens-0012-m1		33658	2272440.00	60448.00	1970232.00	67.52	0.445128

Table 11.13: Comparison of execution time between the baseline solver without machine learning and the solver using the small model with a threshold of 0.5 on the test instances (A: Total AI / Solver without AI total, B: Python AI / # propagation (ms))

Instance	Solver without AI	AI total (ms)	AI Java (ms)	AI Python (ms)	A	B
ColouredQueens-07	102674	25407913.00	121469.00	22714504.00	247.46	0.451725
CostasArray-10	6279	887482.00	7471.00	790122.00	141.34	0.479710
MagicSquare-03-sum	8	57.00	1.00	50.00	7.12	0.446429
Ortholatin-005	324	109642.00	1132.00	97304.00	338.40	0.377045
Subisomorphism-si2-m4Dr6-m625-04	2794	17387.00	1708.00	14283.00	6.22	57.825911
Subisomorphism-si4-m4Dr2-m625-02	729	511.00	132.00	338.00	0.70	112.666667
Sudoku-s03b-alldiff	16	62.00	1.00	55.00	3.88	0.348101
Sudoku-s08c-alldiff	5	135.00	1.00	120.00	27.00	0.372671
Sudoku-s11b-alldiff	3	136.00	1.00	121.00	45.33	0.371166
Sudoku-s11c-alldiff	6	107.00	1.00	94.00	17.83	0.367188
Sudoku-s12a-alldiff	3	321.00	2.00	286.00	107.00	0.393398
Sudoku-s13a-alldiff	5	18.00	0.00	15.00	3.60	0.192308
Sudoku-s13c-alldiff	3	26.00	0.00	24.00	8.67	0.342857
bqwh-15-106-02_X2	8	14071.00	106.00	12513.00	1758.88	0.357606
bqwh-15-106-07_X2	38	36582.00	288.00	32560.00	962.68	0.358812
bqwh-15-106-25_X2	74	19701.00	142.00	17547.00	266.23	0.367623
bqwh-15-106-51_X2	52	16011.00	131.00	14240.00	307.90	0.348652
bqwh-15-106-81_X2	10	3745.00	33.00	3327.00	374.50	0.373526
bqwh-18-141-50_X2	266	385277.00	2593.00	343321.00	1448.41	0.383867
bqwh-18-141-75_X2	106	61342.00	477.00	54510.00	578.70	0.381541
sudoku368_X2	47	53.00	2.00	45.00	1.13	0.517241

Table 11.14: Comparison of execution time between the baseline solver without machine learning and the solver using the random model following the baseline model’s distribution with a threshold of 0.5 on the training instances (A: Total AI / Solver without AI total, B: Python AI / # propagation (ms))

Instance	Solver without AI total (ms)	AI total (ms)	AI Java (ms)	AI Python (ms)	A	B
AllInterval-006	7	86.00	5.00	28.00	12.29	0.013917
AllInterval-009	4612	16077.00	626.00	8068.00	3.49	0.037331
Allergy	4	0.00	0.00	0.00	0.00	0.000000
Alpha	3279	159.00	3.00	117.00	0.05	0.271462
Kakuro-hard-010-sumdiff	16	3.00	0.00	0.00	0.19	0.000000
Kakuro-hard-056-sumdiff	6	21.00	6.00	2.00	3.50	0.006270
Kakuro-hard-062-sumdiff	11	4.00	1.00	1.00	0.36	0.009709
Kakuro-hard-137-sumdiff	13522	36.00	2.00	12.00	0.00	0.015625
Kakuro-medium-035-sumdiff	11	43.00	2.00	12.00	3.91	0.009780
Kakuro-medium-062-sumdiff	7	15.00	1.00	4.00	2.14	0.008529
Kakuro-medium-086-sumdiff	10	4.00	1.00	1.00	0.40	0.010989
Kakuro-medium-129-sumdiff	5349	1840.00	94.00	648.00	0.34	0.014988
KnightTour-06-ext03	10965	583.00	33.00	364.00	0.05	0.105599
Purdey	22	1.00	0.00	0.00	0.05	0.000000
Queens-0009-m1	193	348.00	41.00	98.00	1.80	0.012456
Queens-0012-m1	33658	11737.00	1533.00	3824.00	0.35	0.017890

Table 11.15: Comparison of execution time between the baseline solver without machine learning and the solver using the random model following the baseline model’s distribution with a threshold of 0.5 on the test instances (A: Total AI / Solver without AI total, B: Python AI / # propagation (ms))

Instance	Solver without AI total (ms)			AI total (ms)	AI Java (ms)	AI Python (ms)	A	B
ColouredQueens-07	102674			3158.00	64.00	1096.00	0.03	0.015136
CostasArray-10	6279			2709.00	81.00	1297.00	0.43	0.033784
MagicSquare-03-sum	8			9.00	0.00	3.00	1.12	0.020979
Ortholatin-005	324			134.00	5.00	55.00	0.41	0.019813
Subisomorphism-si2-m4Dr6-m625-04	2794			3065.00	951.00	1705.00	1.10	20.542169
Subisomorphism-si4-m4Dr2-m625-02	729			1648.00	608.00	836.00	2.26	30.962963
Sudoku-s03b-alldiff	16			11.00	1.00	3.00	0.69	0.011673
Sudoku-s08c-alldiff	5			5.00	0.00	1.00	1.00	0.008772
Sudoku-s11b-alldiff	3			5.00	0.00	1.00	1.67	0.009709
Sudoku-s11c-alldiff	6			7.00	1.00	1.00	1.17	0.006024
Sudoku-s12a-alldiff	3			13.00	0.00	4.00	4.33	0.011834
Sudoku-s13a-alldiff	5			2.00	0.00	0.00	0.40	0.000000
Sudoku-s13c-alldiff	3			1.00	0.00	0.00	0.33	0.000000
bqwh-15-106-02_X2	8			7.00	0.00	2.00	0.88	0.014493
bqwh-15-106-07_X2	38			652.00	29.00	200.00	17.16	0.011705
bqwh-15-106-25_X2	74			971.00	41.00	299.00	13.12	0.011591
bqwh-15-106-51_X2	52			396.00	18.00	124.00	7.62	0.012035
bqwh-15-106-81_X2	10			310.00	15.00	97.00	31.00	0.013156
bqwh-18-141-50_X2	266			253.00	14.00	86.00	0.95	0.015122
bqwh-18-141-75_X2	106			265.00	12.00	87.00	2.50	0.013718
sudoku368_X2	47			48.00	2.00	25.00	1.02	0.048077

Table 11.16: Comparison of execution time between the baseline solver without machine learning and the solver using the random model with a uniform distribution and a threshold of 0.5 on the training and test instances (A: Total AI / Solver without AI total, B: Python AI / # propagation (ms))

Instance	Solver without AI total (ms)	AI total (ms)	AI Java (ms)	AI Python (ms)	A	B
AllInterval-006	7	1.00	0.00	0.00	0.14	0.000000
AllInterval-009	4612	1.00	0.00	0.00	0.00	0.000000
Allergy	4	0.00	0.00	0.00	0.00	0.000000
Alpha	3279	2.00	0.00	0.00	0.00	0.000000
ColouredQueens-07	102674	0.00	0.00	0.00	0.00	0.000000
CostasArray-10	6279	1.00	0.00	0.00	0.00	0.000000
Kakuro-hard-010-sumdiff	16	1.00	0.00	0.00	0.06	0.000000
Kakuro-hard-056-sumdiff	6	3.00	2.00	0.00	0.50	0.000000
Kakuro-hard-062-sumdiff	11	1.00	0.00	0.00	0.09	0.000000
Kakuro-hard-137-sumdiff	13522	1.00	0.00	0.00	0.00	0.000000
Kakuro-medium-035-sumdiff	11	1.00	0.00	0.00	0.09	0.000000
Kakuro-medium-062-sumdiff	7	1.00	0.00	0.00	0.14	0.000000
Kakuro-medium-086-sumdiff	10	1.00	1.00	0.00	0.10	0.000000
Kakuro-medium-129-sumdiff	5349	1.00	0.00	0.00	0.00	0.000000
KnightTour-06-ext03	10965	2.00	1.00	0.00	0.00	0.000000
MagicSquare-03-sum	8	0.00	0.00	0.00	0.00	0.000000
Ortholatin-005	324	1.00	0.00	0.00	0.00	0.000000
Purdey	22	0.00	0.00	0.00	0.00	0.000000
Queens-0009-m1	193	0.00	0.00	0.00	0.00	0.000000
Queens-0012-m1	33658	2.00	1.00	0.00	0.00	0.000000
Subisomorphism-si2-m4Dr6-m625-04	2794	173.00	98.00	41.00	0.06	13.666667
Subisomorphism-si4-m4Dr2-m625-02	729	293.00	161.00	49.00	0.40	24.500000
Sudoku-s03b-alldiff	16	1.00	0.00	0.00	0.06	0.000000
Sudoku-s08c-alldiff	5	0.00	0.00	0.00	0.00	0.000000
Sudoku-s11b-alldiff	3	1.00	0.00	0.00	0.33	0.000000
Sudoku-s11c-alldiff	6	1.00	0.00	0.00	0.17	0.000000
Sudoku-s12a-alldiff	3	0.00	0.00	0.00	0.00	0.000000

Continued on next page

Table 11.16: Comparison of execution time between the baseline solver without machine learning and the solver using the random model with a uniform distribution and a threshold of 0.5 on the training and test instances (A: Total AI / Solver without AI total, B: Python AI / # propagation (ms)) (continued)

Instance	Solver without AI total (ms)				AI Python (ms)				A		B	
Sudoku-s13a-alldiff		5		0.00	0.00		0.00	0.00	0.00	0.00	0.000000	0.000000
Sudoku-s13c-alldiff		3		0.00	0.00		0.00	0.00	0.00	0.00	0.000000	0.000000
bqwh-15-106-02_X2		8		0.00	0.00		0.00	0.00	0.00	0.00	0.000000	0.000000
bqwh-15-106-07_X2		38		0.00	0.00		0.00	0.00	0.00	0.00	0.000000	0.000000
bqwh-15-106-25_X2		74		0.00	0.00		0.00	0.00	0.00	0.00	0.000000	0.000000
bqwh-15-106-51_X2		52		1.00	0.00		0.00	0.00	0.02	0.00	0.000000	0.000000
bqwh-15-106-81_X2		10		0.00	0.00		0.00	0.00	0.00	0.00	0.000000	0.000000
bqwh-18-141-50_X2		266		2.00	2.00		1.00	1.00	0.01	0.00	0.000000	0.000000
bqwh-18-141-75_X2		106		0.00	0.00		0.00	0.00	0.00	0.00	0.000000	0.000000
sudoku368_X2		47		2.00	2.00		1.00	1.00	0.04	0.00	0.000000	0.000000

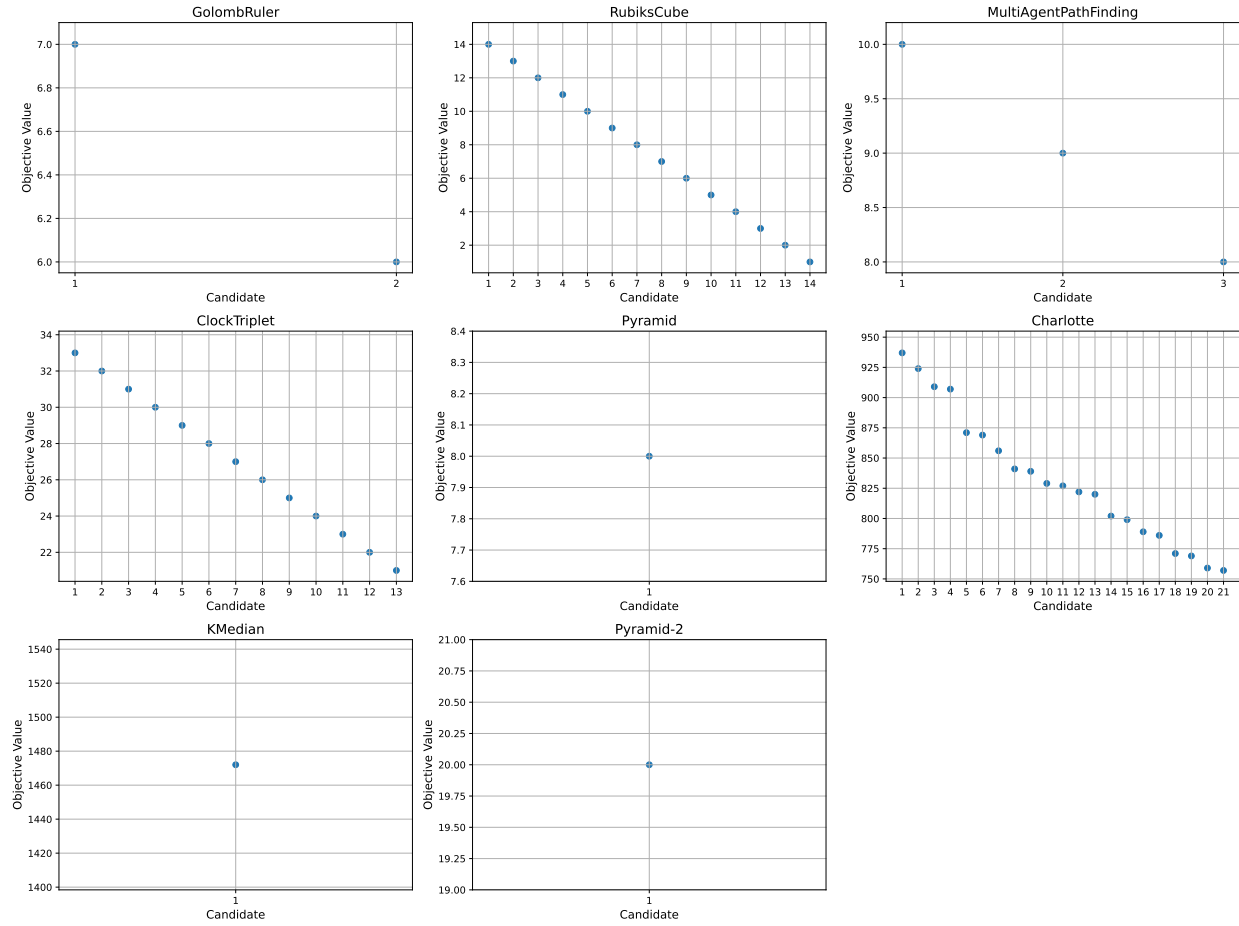


Figure 11.35: Evolution of the objective function value as the different solutions are found, for all instances

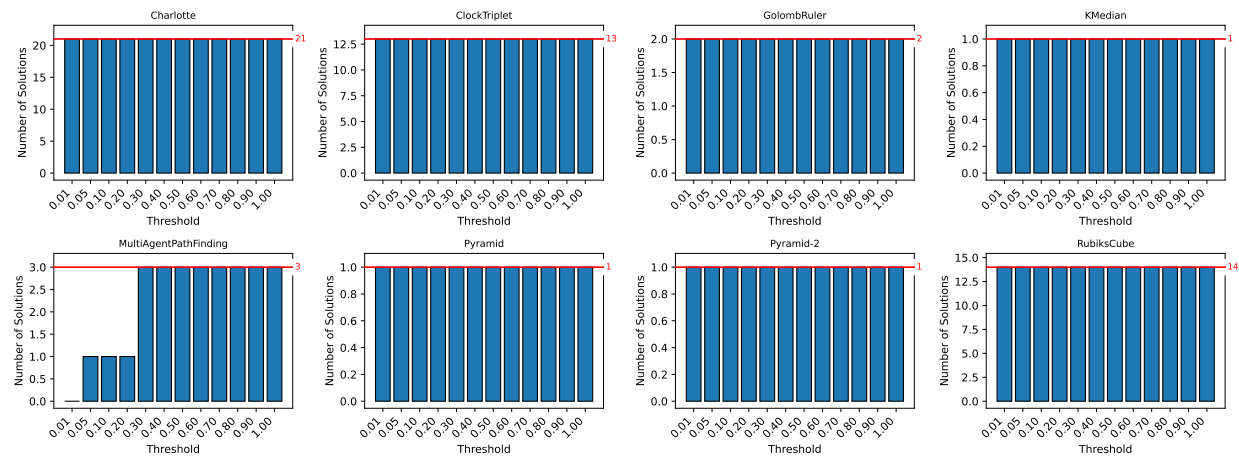


Figure 11.36: Evolution of the number of solutions obtained by the solver using the baseline model as a function of the threshold, in comparison with the classical solver (red line)

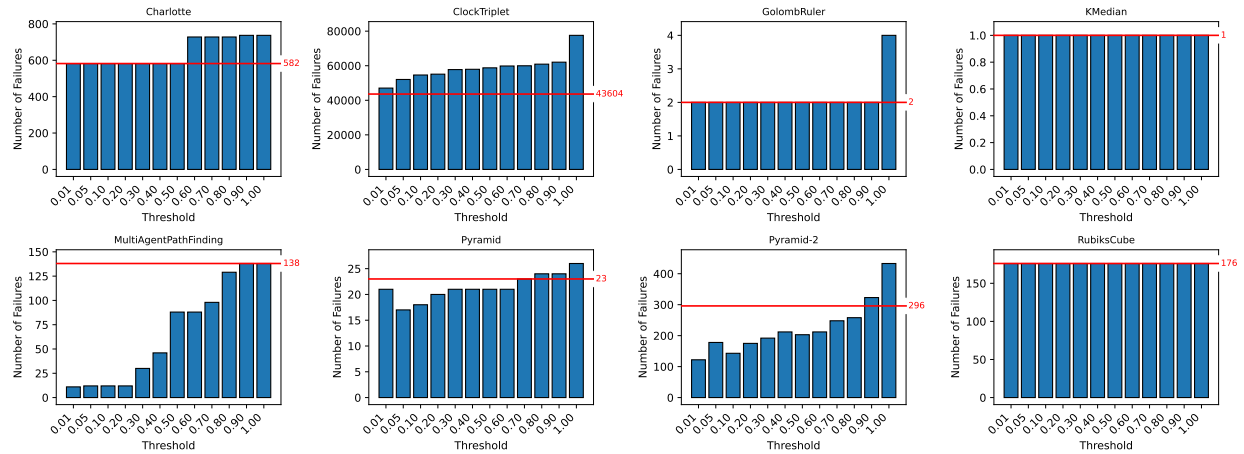


Figure 11.37: Evolution of the number of failures obtained by the solver using the baseline model as a function of the threshold, in comparison with the classical solver (red line)

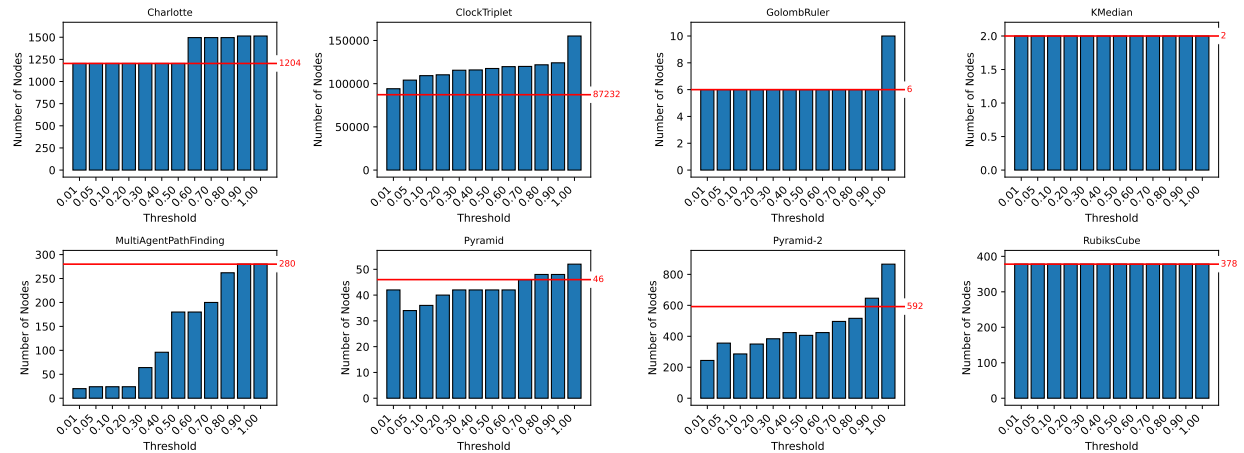


Figure 11.38: Evolution of the number of nodes obtained by the solver using the baseline model as a function of the threshold, in comparison with the classical solver (red line)

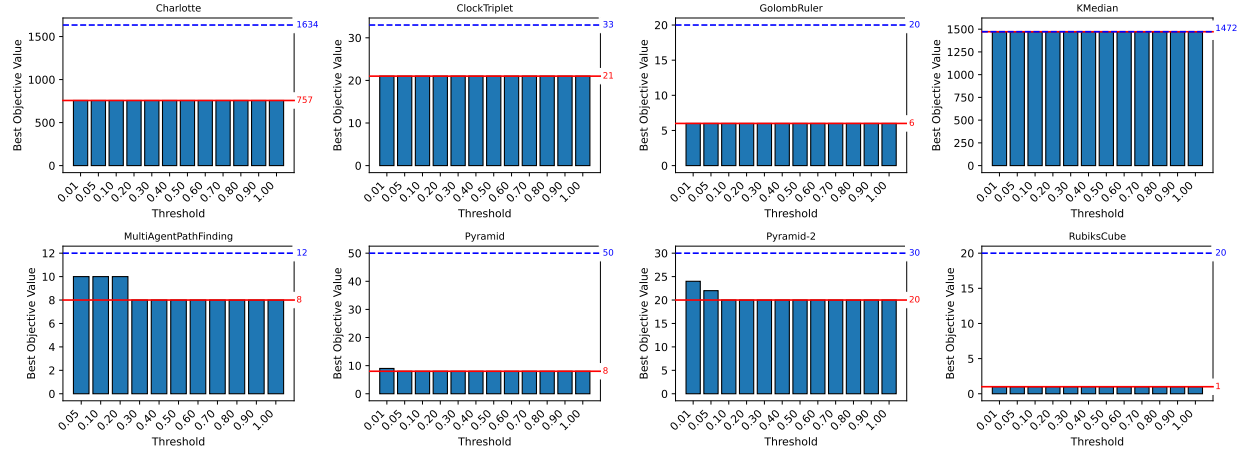


Figure 11.39: Evolution of the objective value of the best solution obtained by the baseline model as a function of the threshold, in comparison with the classical solver (red line)

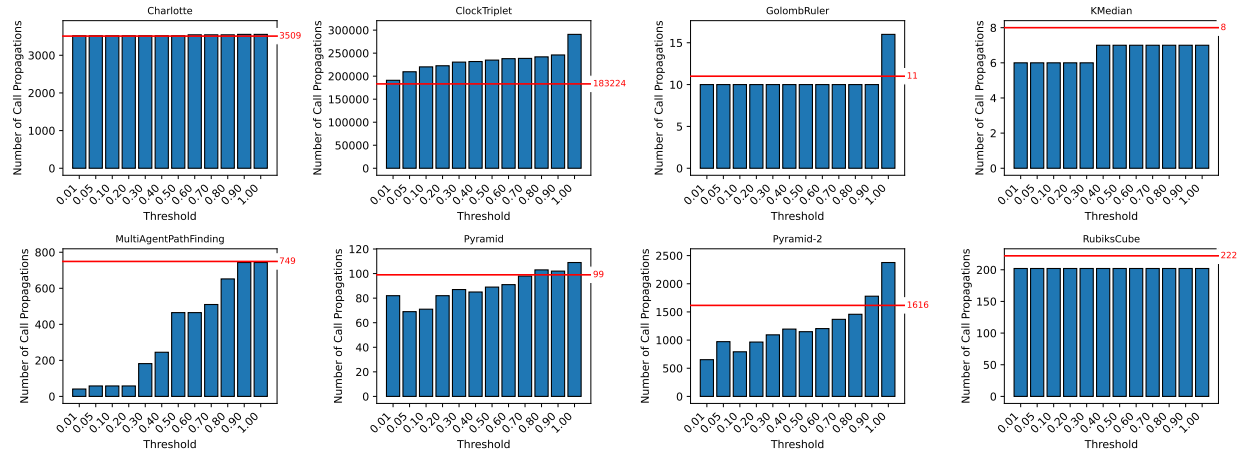


Figure 11.40: Evolution of the number of calls to the *AllDifferent* propagator by the baseline model as a function of the threshold, in comparison with the classical solver (red line)

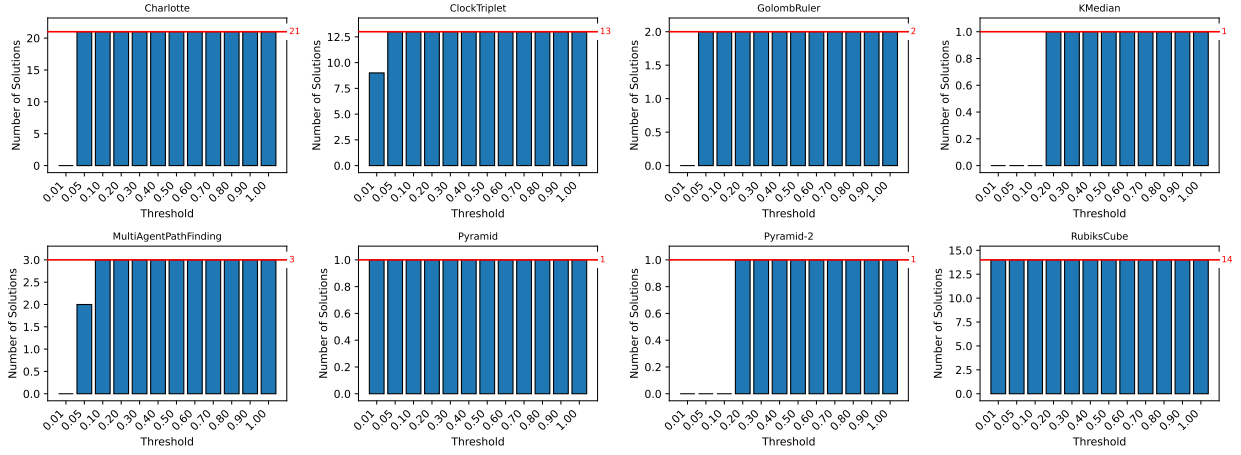


Figure 11.41: Evolution of the number of solutions obtained by the solver using the small model as a function of the threshold, in comparison with the classical solver (red line)

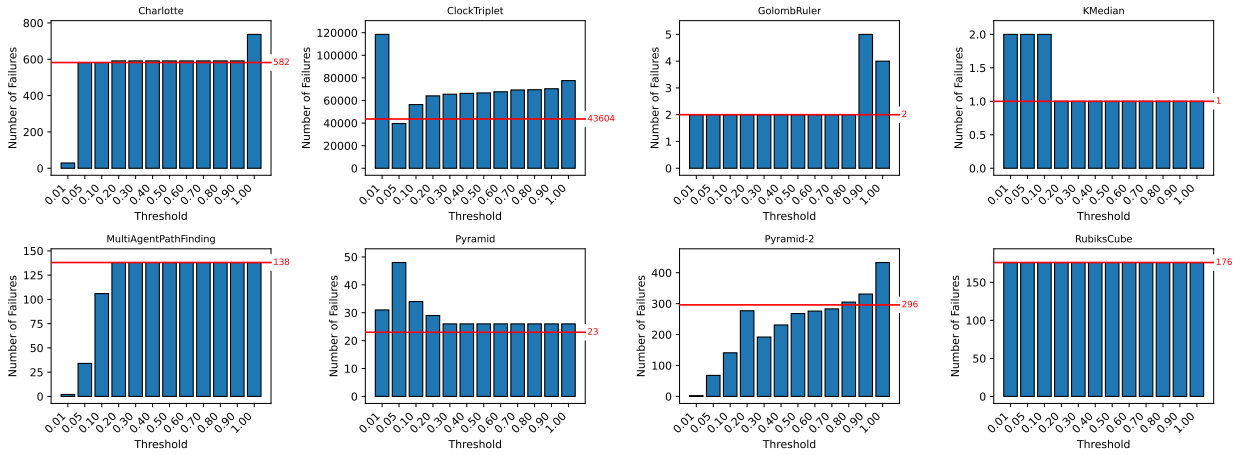


Figure 11.42: Evolution of the number of failures obtained by the solver using the small model as a function of the threshold, in comparison with the classical solver (red line)

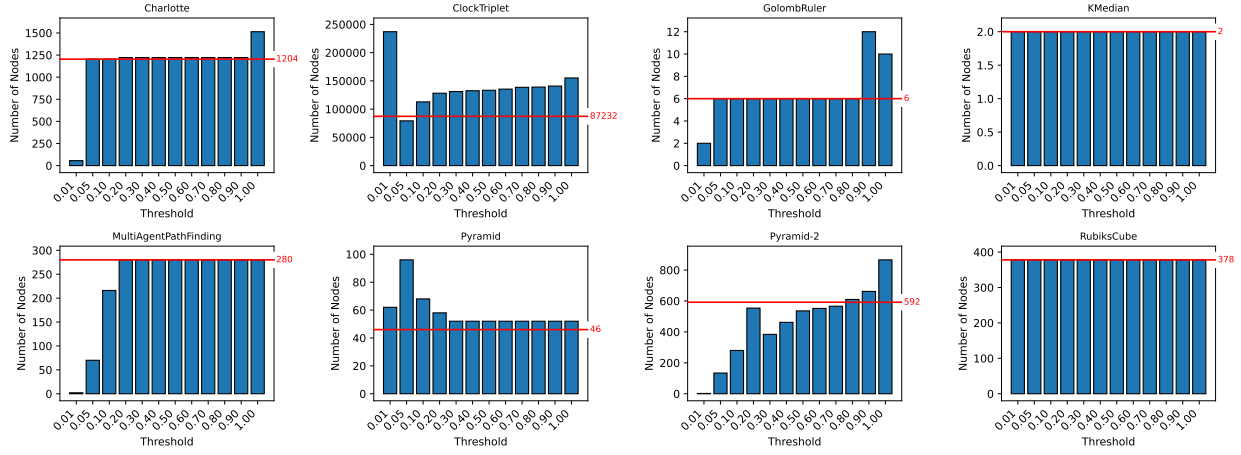


Figure 11.43: Evolution of the number of nodes obtained by the solver using the small model as a function of the threshold, in comparison with the classical solver (red line)

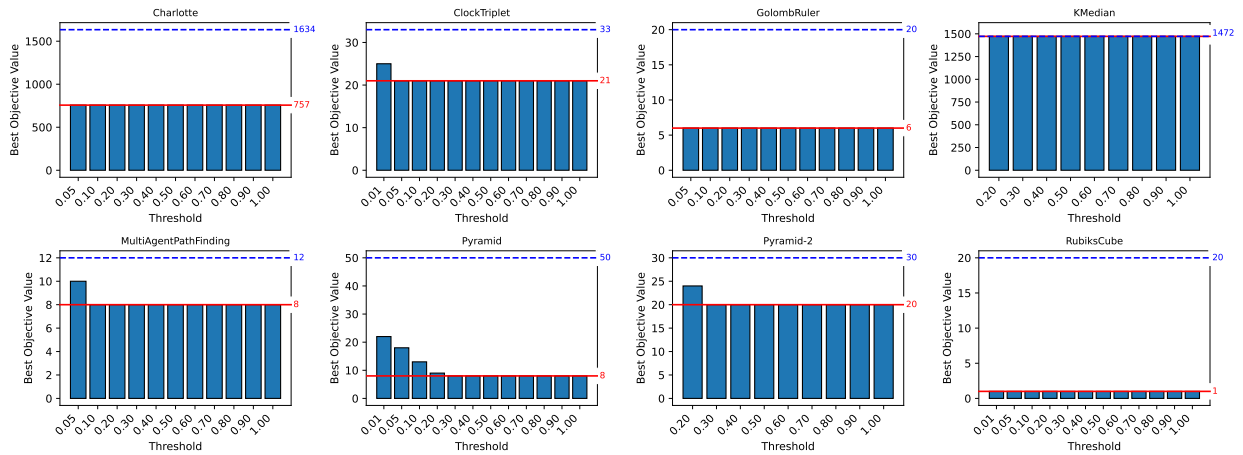


Figure 11.44: Evolution of the objective value of the best solution obtained by the small model as a function of the threshold, in comparison with the classical solver (red line)

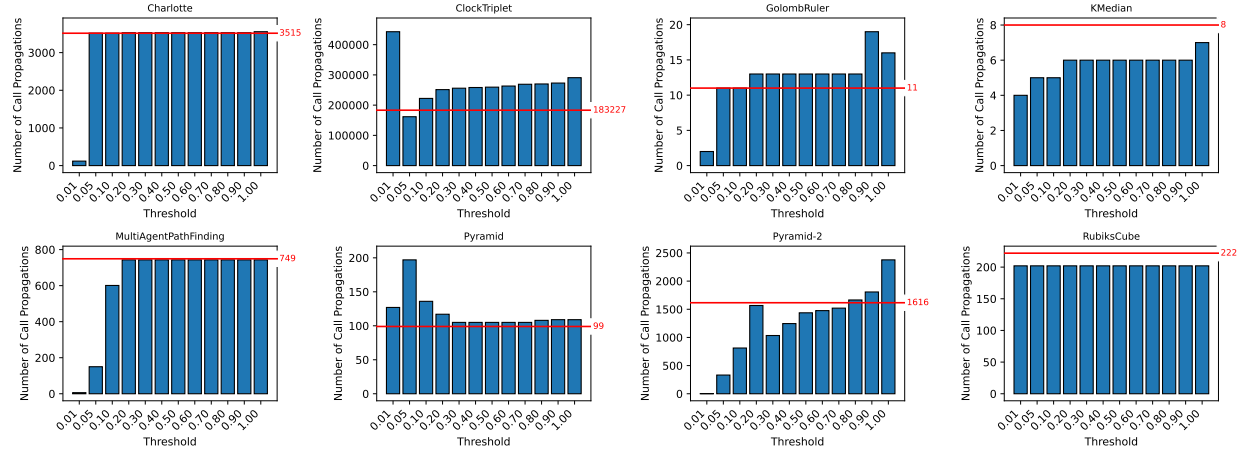


Figure 11.45: Evolution of the number of calls to the *AllDifferent* propagator by the small model as a function of the threshold, in comparison with the classical solver (red line)

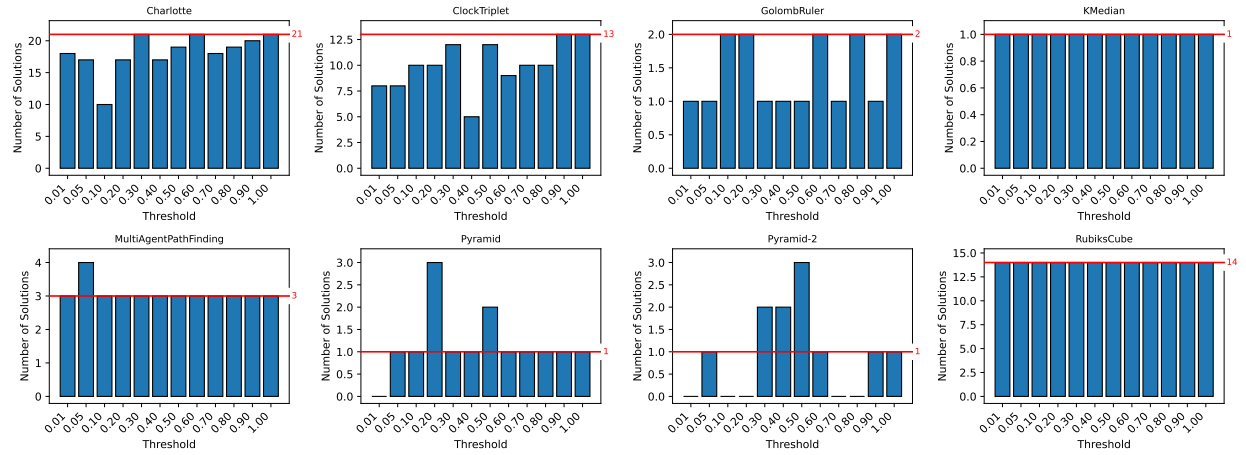


Figure 11.46: Evolution of the number of solutions obtained by the solver using the random model following the baseline model's distribution as a function of the threshold, in comparison with the classical solver (red line)

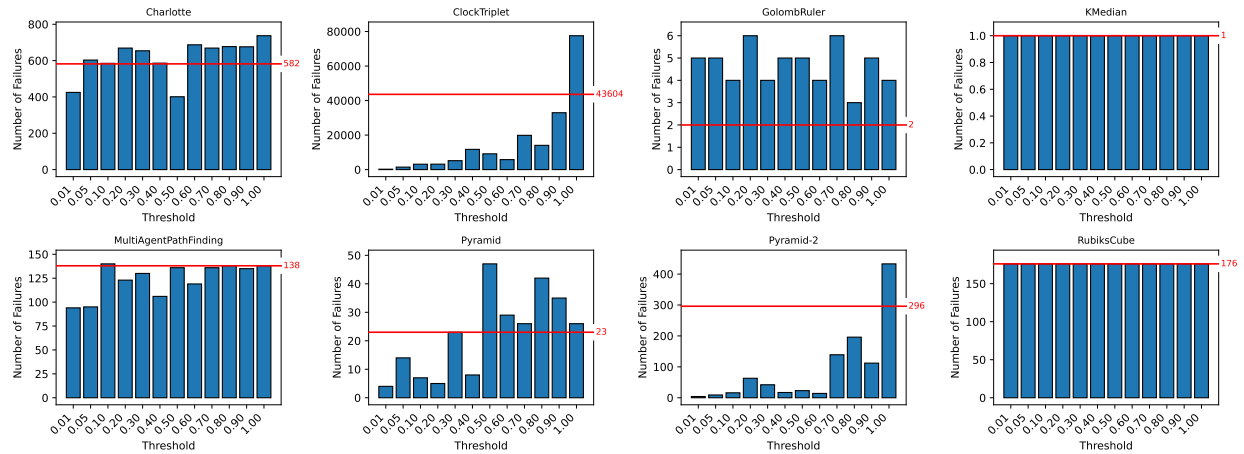


Figure 11.47: Evolution of the number of failures obtained by the solver using the random model following the baseline model's distribution as a function of the threshold, in comparison with the classical solver (red line)

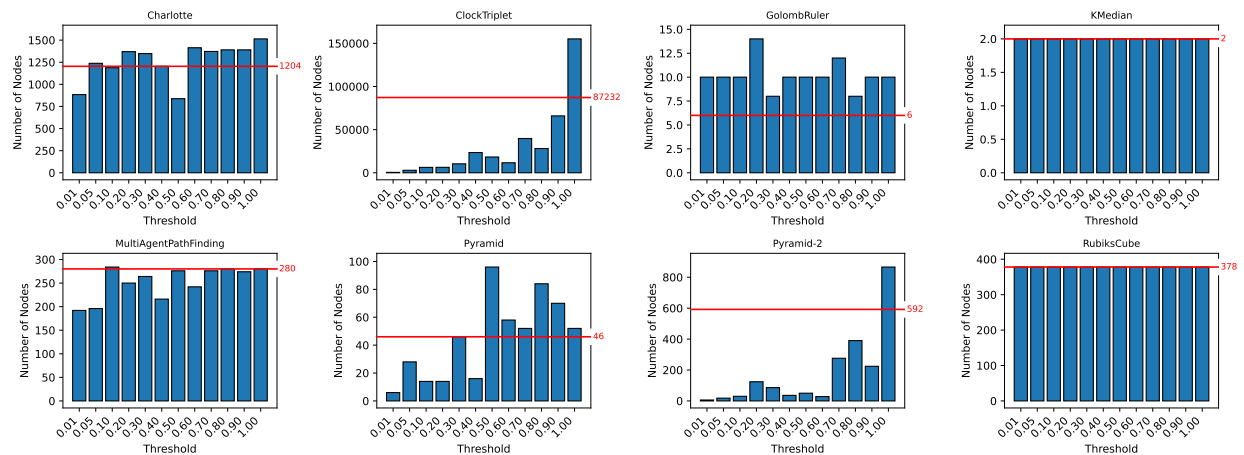


Figure 11.48: Evolution of the number of nodes obtained by the solver using the random model following the baseline model's distribution as a function of the threshold, in comparison with the classical solver (red line)

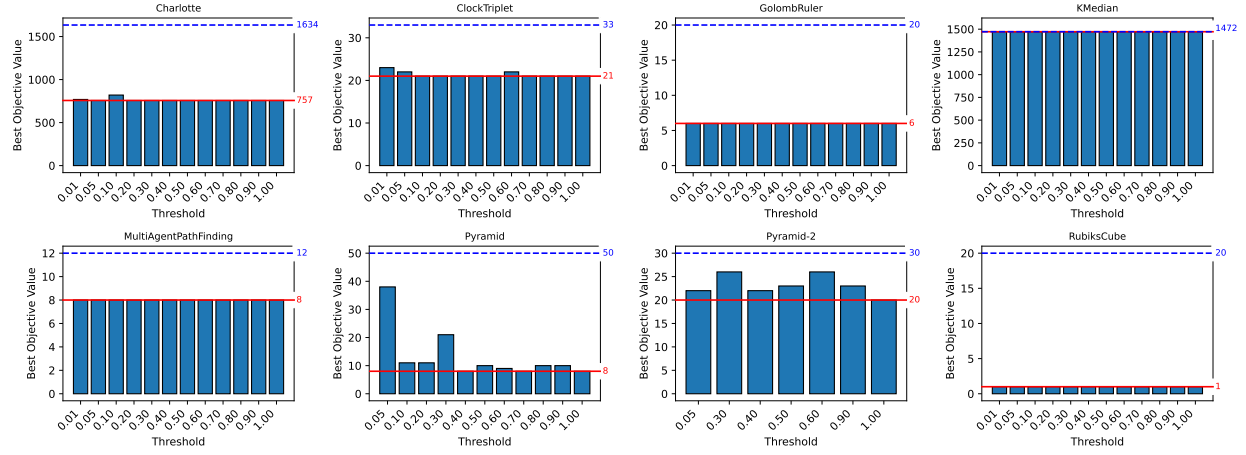


Figure 11.49: Evolution of the objective value of the best solution obtained by the random model following the baseline model's distribution as a function of the threshold, in comparison with the classical solver (red line)

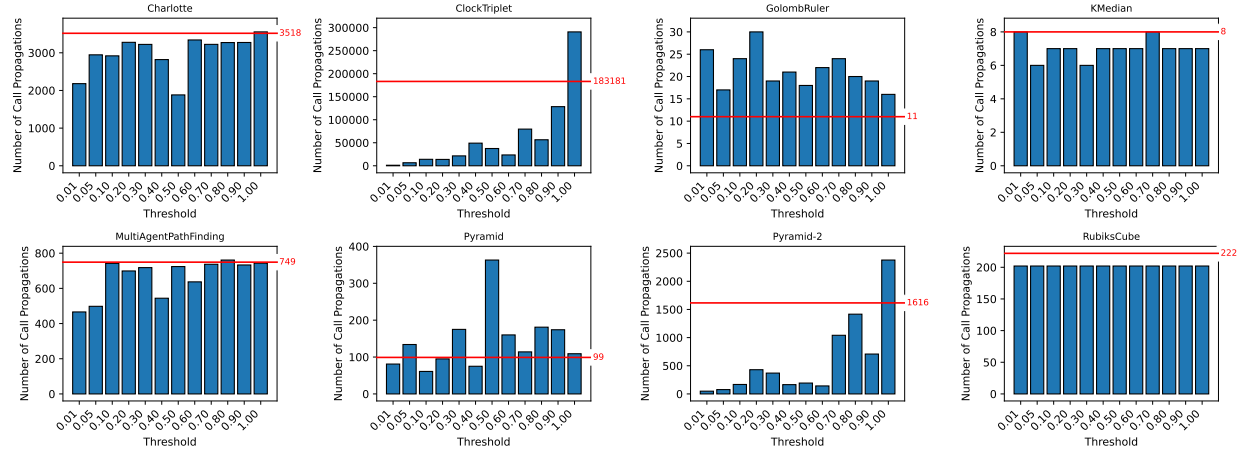


Figure 11.50: Evolution of the number of calls to the *AllDifferent* propagator by the random model following the baseline model's distribution as a function of the threshold, in comparison with the classical solver (red line)

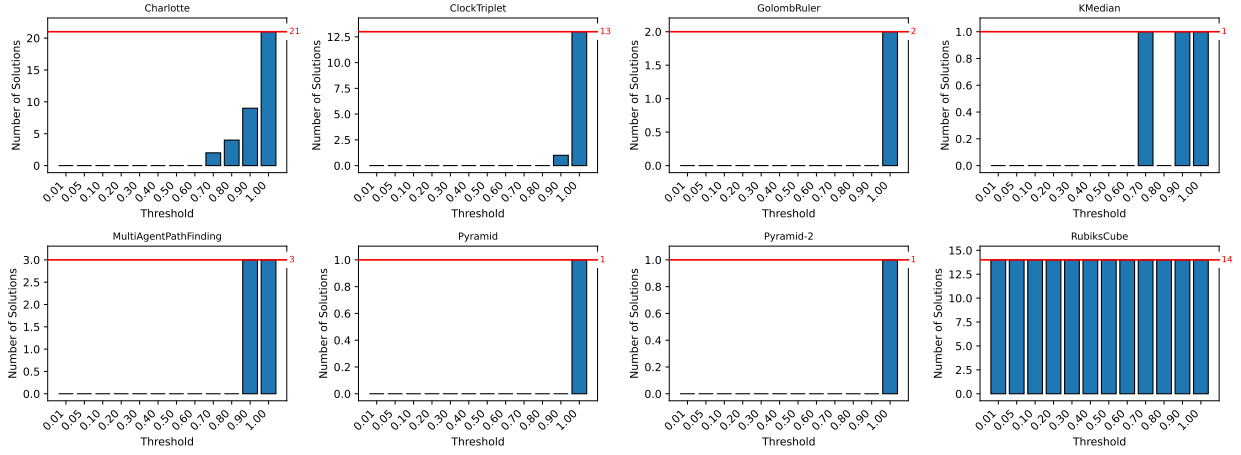


Figure 11.51: Evolution of the number of solutions obtained by the solver using the random model with a uniform distribution as a function of the threshold, in comparison with the classical solver (red line)

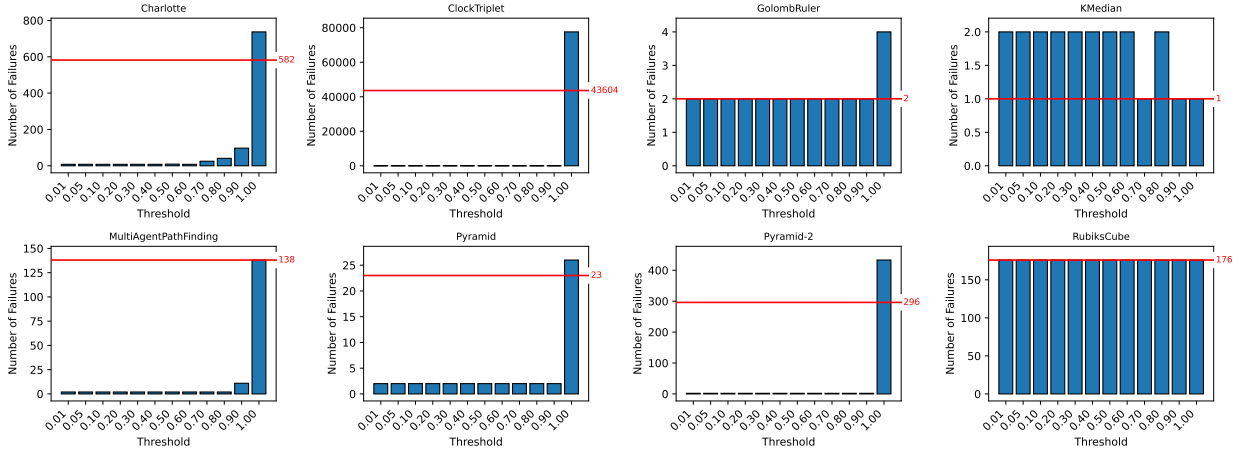


Figure 11.52: Evolution of the number of failures obtained by the solver using the random model with a uniform distribution as a function of the threshold, in comparison with the classical solver (red line)

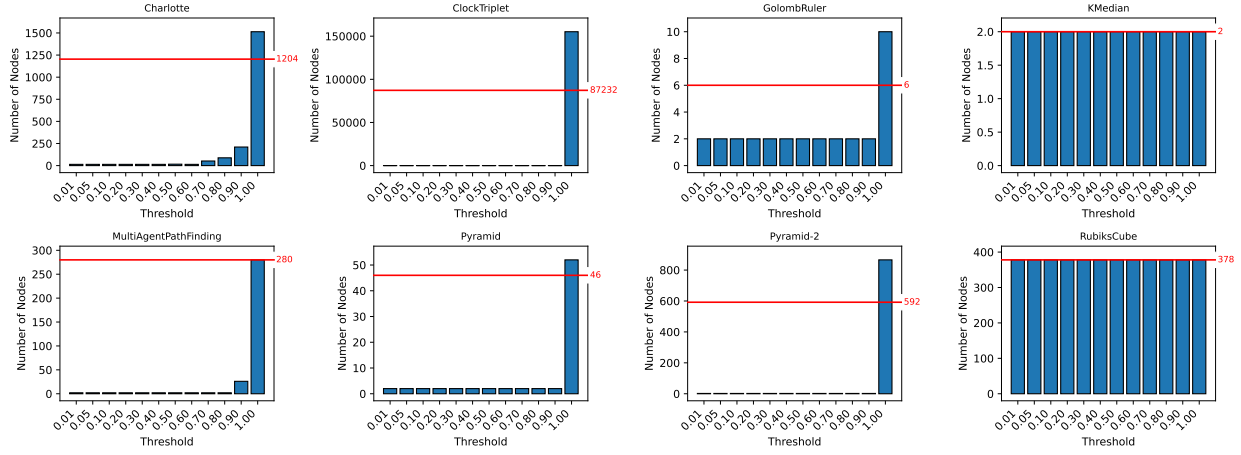


Figure 11.53: Evolution of the number of nodes obtained by the solver using the random model with a uniform distribution as a function of the threshold, in comparison with the classical solver (red line)

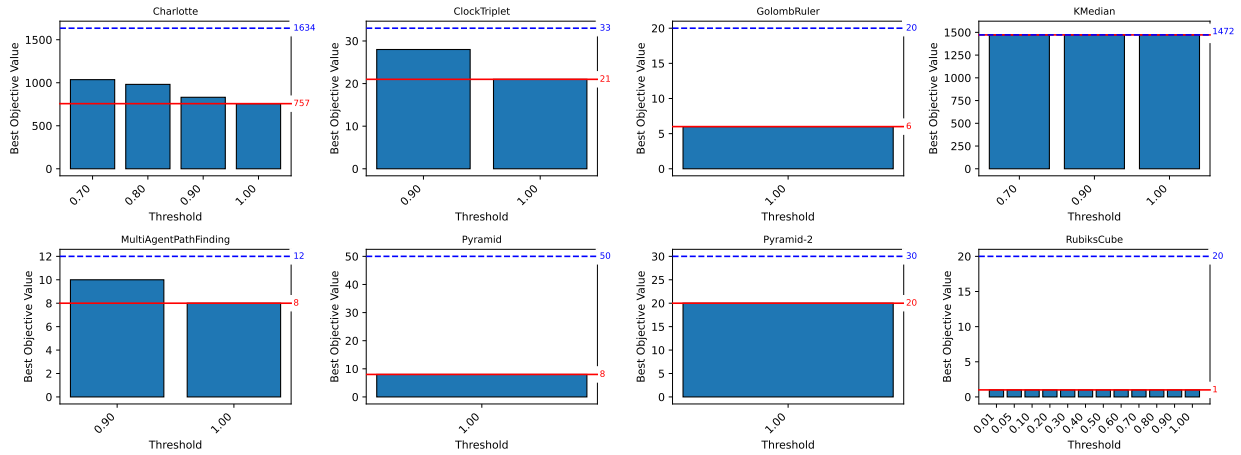


Figure 11.54: Evolution of the objective value of the best solution obtained by the random model with a uniform distribution as a function of the threshold, in comparison with the classical solver (red line)

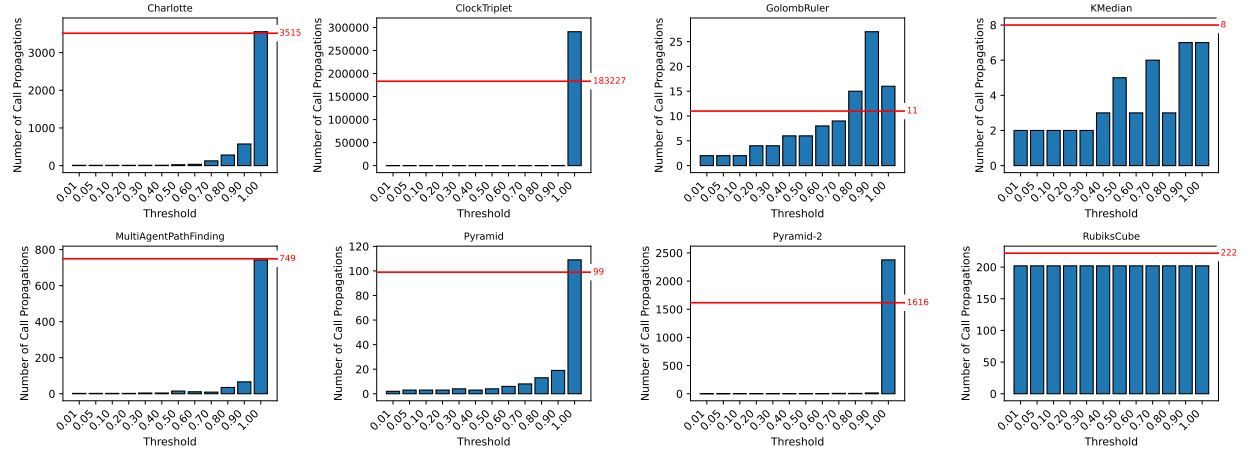


Figure 11.55: Evolution of the number of calls to the *AllDifferent* propagator by the random model with a uniform distribution as a function of the threshold, in comparison with the classical solver (red line)

UNIVERSITÉ CATHOLIQUE DE LOUVAIN
École polytechnique de Louvain

Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | www.uclouvain.be/epl