The Algorithm Selection Problem for Solving Sudoku with Metaheuristics

Danielle Notice
STOR-i Centre for Doctoral Training
Lancaster University
Lancaster, UK
d.a.notice@lancaster.ac.uk

Ahmed Kheiri
Department of Management Science
Lancaster University
Lancaster, UK
a.kheiri@lancaster.ac.uk

Nicos G. Pavlidis
Department of Management Science
Lancaster University
Lancaster, UK
n.pavlidis@lancaster.ac.uk

***Abstract*—In this paper we study the algorithm selection problem and instance space analysis for solving Sudoku puzzles with metaheuristic algorithms. We formulate Sudoku as a combinatorial optimisation problem and implement four local-search metaheuristics to solve the problem instances. A feature space is constructed and instance space analysis (ISA) methodology is applied to this problem for the first time. The aim is to use ISA to determine how these features affect the performance of the algorithms and how they can be used for automated algorithm selection. We also consider algorithm selection using multinomial logistic regression models with l_1-penalty for comparison. Different algorithm performance metrics are considered and we found that the choice of these metrics affected whether the use of ISA was worthwhile. We found that when the performance of metaheuristic solvers is similar, predicting whether an algorithm will perform "well" is also useful.**

***Index Terms*—algorithm selection problem, metaheuristics, instance space analysis, Sudoku.**

I. Introduction

Sudoku is a Japanese puzzle which consists of an $n^2 \times n^2$ grid divided into n^2 sub-grids each of size $n \times n$. n is the *order* of the puzzle, with $n = 3$ being the most popular (see Fig. 1). The objective is to fill each cell in a way that every row, column and sub-grid contains each integer between 1 and n^2 inclusive exactly once.

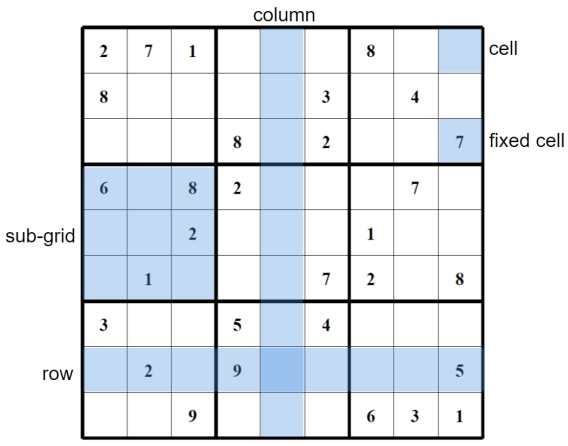

Fig. 1. Sample board of a 9×9 Sudoku puzzle

Sudoku puzzles are a special case of *Latin squares* - grids of equal dimensions in which every symbol occurs exactly once in every row and every column [1]. Constructing or completing a Latin square or a Sudoku puzzle from a partially filled grid are both $\mathcal{NP}$-complete problems. The solution space for an empty 9×9 Sudoku grid contains approximately 6.67×10^{21} possible combinations. However, the pre-filled cells serve as constraints and reduce the number of possible combinations [2]. There is no deterministic algorithm which can solve all possible Sudoku problem instances in polynomial time, so heuristic methods are popular [1], [2]. This is often the case for combinatorial optimisation (CO) problems, which are a class of problems whose solution is in a finite or countably infinite set [3]. Although many CO problems have been widely studied, there is ongoing research about which algorithm performs best on a particular instance, or class of instances. The No Free Lunch Theorem [4] warns that there is no single algorithm that will be guaranteed to perform well across all problem instances. When trying to determine which algorithm is most suitable for a set of instances, exhaustive testing may be impractical and expensive. It is therefore important to understand the relationship between instances and algorithm performance. This can then be used for an automated algorithm selection model to predict which algorithm in a portfolio is most suitable for solving a given instance.

A useful framework for selecting an algorithm which will perform best for a collection of problem instances is presented in [5]. The **algorithm selection problem**, depicted in Fig. 2, has four components which make up the **meta-data**:

- the ***problem space*** $\mathcal{P}$ - all the relevant instances of a problem in an application domain;
- the ***feature space*** $\mathcal{F}$ - a set of features extracted to characterise the problem space;
- the ***algorithm space*** $\mathcal{A}$ - a set of algorithms available to solve all instances in $\mathcal{P}$;
- the ***performance space*** $\mathcal{Y}$ - a set of performance metrics for each of the algorithms $\alpha \in \mathcal{A}$ solving each problem instance $x \in \mathcal{P}$.

The authors in [6] define the algorithm selection problem as: for a given problem instance $x \in \mathcal{P}$, with feature vector $f(x) \in \mathcal{F}$, find the selection mapping $S(f(x))$ into algorithm

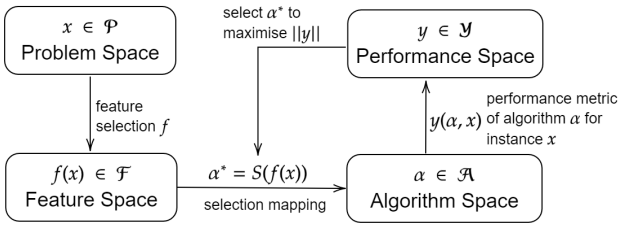

Fig. 2. Framework for the Algorithm Selection Problem [6]

space $\mathcal{A}$, such that the selected algorithm $\alpha \in \mathcal{A}$ maximises the performance metric $\|y\|$ for $y(\alpha, x) \in \mathcal{Y}$. Under this framework, the problem instances are first translated to some feature space. This step involves the construction of suitable features for the problem, and then the selection of a useful subset of these features. A prediction model is then trained on the existing instances for which algorithm performs best.

In order to train a useful prediction model, it is necessary that: (i) the meta-data for the algorithm selection problem comprise of a large set of diverse instances; (ii) suitable features which characterise the instances are chosen; and (iii) a diverse portfolio of algorithms and performance metrics are considered. Problem instances, their features and algorithm performance can be represented in a shared space, referred to as the ***instance space*** [7].

An overview of research in which metaheuristic algorithms have been applied to solve Sudoku puzzles by modifying the general template of the algorithms is presented in [8]. The performance of these metaheuristics has been evaluated using different criteria such as the rate of success, the number of iterations and the amount of time needed to solve the puzzle. However, there is no set of benchmark instances or common performance criteria across studies. For these reasons, in this study we evaluate the performance of four metaheuristic solvers for Sudoku, attempting to apply automated algorithm selection. As part of this process we first construct a feature space for Sudoku puzzles, which is an issue that has received relatively little attention. The meta-data created are then analysed using the Melbourne Algorithm Test Instance Library with Data Analytics (MATILDA) tool [9] and the results are discussed.

II. Problem Setting

A. Problem Instances

The instance space included 1000 problem instances, all of order $n = 3$. These were selected from five sources which were collated in the *Tdoku* GitHub repository [10]. These are all *proper* puzzles, meaning that the solution is unique. The combined dataset, after removing duplicate instances, contains approximately 3 million problem instances.

To select a subset of instances to conduct the analysis, 20 features related to the structure of the puzzle were extracted. These features, which are explained in detail in Section II-C, describe the location of the fixed cells and the possible values that empty cells could take. We selected these 20 features because they are among the cheapest (in terms of computational cost) Sudoku features to calculate. Principal component analysis (PCA) was applied on this feature space and the first two principal components were used to cluster the problem instances using k-means clustering. 1000 clusters were created and one problem instance was randomly selected from each cluster to construct a diverse set of problem instances.

B. Algorithm Space

We considered four local-search metaheuristic solvers: **simulated annealing (SA)** [11], **record-to-record travel (RR)** [12], **reduced variable neighbourhood search (RVNS)** [13] and **steepest descent algorithm (SD)**. For all of the algorithms, a solution is represented as a 9×9 matrix $\mathbf{X}$ where x_{ij} is the digit in cell (i, j). To initialise the solution, for each cell, a random number is selected from a list of numbers that include all the numbers that could be assigned to the cell without violating any of the fixed cell constraints. This is done in such a way that the sub-grid constraints are satisfied. The cost function represents the number of values from one to nine that are not present in each row, each column and each sub-grid. A problem instance is **solved** when the cost is zero.

SA accepts a neighbouring solution $\mathbf{X}'$ of $\mathbf{X}$ if it improves the objective value, or is otherwise accepted with probability $e^{-\Delta/T}$, where $\Delta = f(\mathbf{X}') - f(\mathbf{X})$ is the deterioration of the objective value if the solution is accepted and T is a control parameter, called temperature. Neighbouring solutions were found using the *Global Swap* operator, where the values of two unfixed cells are exchanged. We implemented the SA solver as described in [1]. They use a geometric cooling schedule, $T_{k+1} = \alpha T_k$, where $\alpha \in (0, 1)$. In their method, the algorithm generates a series of candidate solutions at each value T_k. The length of each series depends on the size of the problem instance. The initial temperature T_0 is determined by performing a number of swaps and calculating the standard deviation of the costs of the solutions found. Finally, a "reheat" mechanism is employed, where the algorithm is restarted at T_0 if no improvement is made to the solution for a fixed number of solution series.

RR accepts a solution which has a cost value no worse than σ from the best solution found so far, where σ is a fixed deviation value. The Global Swap operator was also used. SD accepts a solution which improves or maintains the cost value of the current solution. The *Swap* operator was used to find neighbouring solutions, where the values of two unfixed cells in the same sub-grid are exchanged.

RVNS considers multiple neighbourhood structures. Given a solution, a random point from a given neighbourhood is selected and accepted if it improves the current solution. Otherwise, another neighbourhood structure is considered. Along with the Swap and Global Swap operators, the neighbourhood operators we considered are described in [8] and listed below:

- *Insert* - the value of a chosen cell in a sub-grid is inserted in front of another chosen cell.

- *CPOEx* - a cell between the second and sixth cell in a sub-grid is selected as the centre point to find exchange pairs. Values for pair of cells, each equidistant from the centre, are swapped until at least one cell in the pair is fixed.
- *Invert* - two cells in the sub-grid are selected and the order of the sub-sequence of cells between them is reversed.

Parameters for SA and RR were tuned using a popular set of benchmark puzzles [14]. After some experiments, for SA, the number of series before reheat was set to 20 and $\alpha = 0.9$, and $\sigma = 1$ in RR. RVNS and SD have no parameters to tune.

C. Feature Space

The feature space for Sudoku has not been widely studied [1], [15]. Several of the insights are based on the discussions of Sudoku enthusiasts and need to be formally tested.

1) Features from Puzzle Mask: These are features related to the structure of the puzzle, such as the puzzle order and the number of fixed cells. However, the difficulty of the puzzle largely depends on the configuration of the fixed cells, which is called the **puzzle mask** [15]. In [15] it is shown (by example) that puzzles with the same mask but different digits can have different difficulties. Thus we consider the following features:

- number of fixed cells;
- total number of empty rows, columns and sub-grids;
- distribution statistics of fixed digits per sub-grid (mean, variation coefficient, entropy, minimum and maximum).

2) Features from Puzzle Rating Systems: Several rating systems used by puzzle creators consider some measure of how a logic-based algorithm solves the problem. An online Sudoku platform [16] considers how many different logic techniques it takes to complete a puzzle, and how many times each technique is used. The techniques are each given a score based on how advanced they are, and the solver tries to apply them in order of difficulty. Such rating require solving the puzzle which is computationally very expensive. However, certain features related to some of the simpler strategies can be extracted from a puzzle with little computational effort. The work in [17] explains several common approaches used by humans when solving puzzles, which have been implemented in logic-based solvers. The simplest of these strategies is to find *naked singles*, which are cells that can only take a single value when the filled cells in the same row, column and sub-grid are considered. This strategy can also be extended to two or three candidate values for a cell (*naked pairs or triples*). We include the following features:

- the number of naked singles, pairs and triples apparent on an empty puzzle grid;
- the distribution statistics for the number of cells that can take a particular value.

3) Features of GCP Formulation: Since Sudoku is comparable to other CO problems [18], the next consideration was to reformulate the puzzle to a type of problem with a set of well studied features. One such problems is the **graph colouring problem (GCP)**. R. Lewis [19] demonstrates that Sudoku can be considered a special case of the *partial Latin square problem* in which the constraint of appropriately filling out the sub-grids must also be satisfied. This is a precolouring problem for which a subset of the vertices has already been assigned colours. The precolouring problem can be converted to a standard GCP using graph contraction operations. Transforming the problem instances, we extract the following features which are used in [20]:

- the number of vertices and edges;
- the density of the graph;
- the vertex degree (mean and standard deviation);
- the average length of the shortest paths for all possible vertex pairs;
- betweenness centrality (mean and standard deviation) - fraction of all shortest paths connecting all pairs of vertices that pass through a given vertex;
- the graph clustering coefficient;
- the Weiner number.

4) Features of SAT Reformulation: Sudoku can be formulated as a constraint satisfaction **(SAT) problem**, and for SAT a large set of features have been studied and found to be related to instance difficulty [21]. The SAT problem can be summarised as follows: given a set of clauses, determine if there exists a truth assignment that satisfies all of them simultaneously. The work in [22] encodes the Sudoku puzzle into conjunctive normal form (CNF) using the minimal encoding which suffices to characterise the puzzle. For each entry in the $N \times N$ grid $\mathcal{S}$, we associate N variables using the notation s_{xyz} to refer to variables. Variable s_{xyz} is assigned true if and only if the entry in row x and column y is assigned value z. The constraints to be added to our SAT encoding refer to each entry, each row, each column and each sub-grid. Fixed cells are represented by unit clauses, which have a single literal which must be assigned true. The authors in [21] present a set of features for constructing predictive models for the difficulty of SAT problems. They include features of different graph representations of a SAT instance. The *clause graph* has nodes representing clauses and an edge between two clauses whenever they share a negated literal and may be a useful way to characterise the configuration of the fixed cells. The *variable graph* has nodes representing variables and an edge between variables that occur together in at least one clause. Another group of features are from solving the linear programming relaxation of an integer programming representation of the SAT instance. We extract the following features:

- Problem Size Features - number of clauses, variables, and ratios of both;
- Clause Graph - nodes degree statistics (mean, variation coefficient and entropy), weighted clustering coefficient statistics;
- Variable Graph - nodes degree statistics;
- LP-relaxation Features - objective value, fraction of variables set to 0 or 1, variable integer slack statistics.

After completely removing uninformative features (those with the same value for all instances), we have a set of

54 features. It is possible that some features from these reformulations will provide redundant information. The feature selection algorithm will resolve this issue.

D. Performance Space

For each problem instance, 20 initial solutions were generated. Every metaheuristic solver was initialised with each of these 20 solutions and executed until either a cost of zero was reached (the problem instance was solved), or a maximum 5×10^5 cost function evaluations was performed. For a given instance, the **success rate** is the proportion of runs in which a solution with cost zero is found within the fixed budget.

We also use a performance metric which considers both the objective function value and time. For a given instance and run, **cost-time** is defined as:

$$c_{\text{best}} + \frac{i}{\text{maxIts}}, \tag{1}$$

where c_{best} is the lowest cost achieved during a run, and i is the earliest iteration in which c_{best} is achieved. We considered the **mean cost-time** across the 20 runs.

III. Method

The Melbourne Algorithm Test Instance Library with Data Analytics (MATILDA) [9] is a tool that follows Rice's algorithm selection framework, and extends it to enable visualisation and measurement of algorithm footprints in a process called Instance Space Analysis (ISA) [7]. The ISA methodology, which is implemented in MATLAB [23] and available online in MATILDA [9], consists of several algorithms, four of which we outline in Sections III-A to III-D.

In the following, the ***feature matrix*** $\mathbf{F} \in \mathbb{R}^{m \times n}$ stores in its columns the m features that describe each of the n problem instances. Similarly, the ***performance matrix*** $\mathbf{Y} \in \mathbb{R}^{a \times n}$ stores in its columns the performance of each of the a algorithms on each problem instance.

A. Preparation for Learning of Instance Meta-data

First, MATILDA pre-processes the feature matrix $\mathbf{F}$ and the performance matrix $\mathbf{Y}$ to construct an instance space. A binary performance metric, $\mathbf{Y_{bin}}$, is calculated from $\mathbf{Y}$ based on a user-defined threshold for "good" performance. This metric can either be absolute (compared to a fixed threshold) or relative (compared to the performance of the best algorithm for a given instance). Both the feature matrix $\mathbf{F}$ and the performance matrix $\mathbf{Y}$ can be normalised by applying a Box-Cox transformation and then standardised to have mean zero and standard deviation one [7].

B. Selection of Instance Features to Explain Difficulty

This algorithm, abbreviated as SIFTED, identifies the features which are most correlated with algorithm performance and are uncorrelated with each other. SIFTED first calculates the absolute value of Pearson's correlation coefficient between the features and algorithm performance. The feature most correlated to the performance metric for each algorithm is selected, as well as any other features moderately correlated ($|\rho| > 0.3$) to the performance of at least one algorithm. To obtain the user-specified number of k features, SIFTED applies k-means clustering to detect groups of similar features. Multiple subsets of k features are obtained by randomly selecting a single feature from each of the k clusters. For each such subset of k features SIFTED applies PCA to reduce the data to two dimensions. Each of the resulting datasets is used to train a Random Forest that predicts $\mathbf{Y_{bin}}$. The optimal subset of features is the one which results in the lowest predictive error [7].

C. Projecting Instances with Linearly Observable Trends

This algorithm, abbreviated as PILOT, aims to find a lower-dimensional projection of the features, $\mathbf{Z} = \mathbf{A_r F}$, such that $\mathbf{Z}$ has a linear relationship with the original features, and with the performance of the different algorithms [24]. This is formulated as minimising the sum of squared approximation errors as:

$$\min_{\mathbf{A_r}, \mathbf{B_r}, \mathbf{C_r}} \quad ||\mathbf{F} - \hat{\mathbf{F}}||_F^2 + ||\mathbf{Y} - \hat{\mathbf{Y}}||_F^2 \tag{2}$$

$$\text{s.t.} \quad \mathbf{Z} = \mathbf{A_r F} \tag{3}$$

$$\hat{\mathbf{F}} = \mathbf{B_r Z} \tag{4}$$

$$\hat{\mathbf{Y}} = \mathbf{C_r Z}, \tag{5}$$

where $\mathbf{A_r} \in \mathbb{R}^{2 \times m}, \mathbf{B_r} \in \mathbb{R}^{m \times 2}, \mathbf{C_r} \in \mathbb{R}^{a \times 2}$. This optimisation problem is solved numerically using the Broyden–Fletcher– Goldfarb– Shanno (BFGS) optimisation algorithm in the MATILDA toolkit [9]. The optimal solution has the highest Pearson correlation coefficient between the distances in the feature space and the distances in the new projection space. The projections, $\mathbf{Z}$, can be used for visualisation.

D. Performance Prediction & Automated Algorithm Selection

In the final stage MATILDA trains a support vector machine (SVM) to predict the binary performance measure, $\mathbf{Y_{bin}}$, for each algorithm using as input the projected features, $\mathbf{Z}$ [7]. The SVMs can be trained with either polynomial or Gaussian kernels. The SVM parameters are tuned either through Bayesian optimisation, or a random search algorithm, both using k-fold stratified cross-validation. If for a given problem instance, there is only one algorithm with good performance, it is selected as the best algorithm for that instance. If multiple algorithms are predicted to perform well, then the algorithm whose model has the highest precision (across all n problem instances) is selected as the best. If none of the algorithms are predicted to be good, then the algorithm with the highest average performance is selected.

IV. Computational Results

All four metaheuristics where implemented in Python version 3.9.10. The results were generated on an Intel(R) Xeon(R) Gold 6248R CPU at 2.6 GHz. The code used to generate the results discussed in this paper can be found at [25].

TABLE I
PROBABILITY DISTRIBUTION OF SUCCESS RATES

	SA	RR	RVNS	SD
Average SR	0.088	0.086	0.062	0.058
Pr(SR $>$ 0)	0.438	0.442	0.345	0.326
Pr(SR $>$ 0.5)	0.041	0.041	0.041	0.039
Pr(SR $=$ 1)	0.041	0.038	0.022	0.011

TABLE II
PROBABILITY DISTRIBUTION OF MEAN COST-TIME

	SA	RR	RVNS	SD
Average CT	2.202	2.085	2.679	2.775
Pr(CT $<$ 2.5)	0.873	0.962	0.275	0.197
Pr(CT $<$ 2.0)	0.127	0.162	0.046	0.044
Pr(CT $<$ 1.0)	0.041	0.041	0.036	0.036

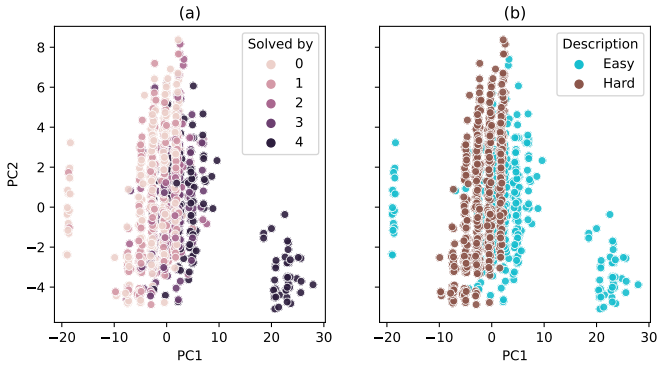

Fig. 3. Visualisation of instances using the 1st and 2nd principal components of the full feature space showing (a) the number of algorithms able to solve the instance at least once and (b) the classification based on logic solvers

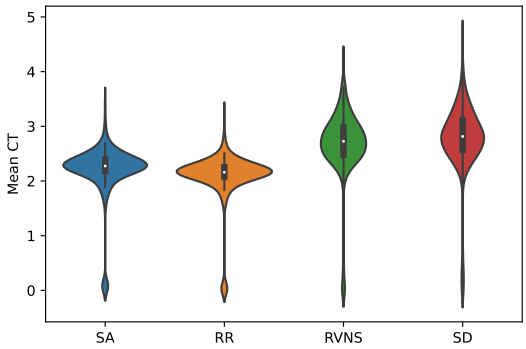

Fig. 4. Distribution of mean cost-time for each algorithm

A. Exploratory Data Analysis

1) Success Rate (SR): Table I shows that for approximately two-fifths of the problem instances SA and RR found the solution with cost equal to zero (solved the instance) at least once in the 20 runs. RVNS and SD found that solution at least once in approximately one-third of the problem instances. 654 of the instances were solved by at least one of the algorithms, while 175 of the instances were solved by all of them. Fig. 3 shows the number of algorithms that were able to solve a given instance at least once. Generally, the instances that three or four algorithms found solvable correspond to data sources which were described as "easy". However there are also instances such as those furthest left which are described as easy but none of the considered algorithms can solve them. This can be explained by the fact that logic-based solvers employ search techniques that are markedly different from those employed by metaheuristic solvers.

2) Mean Cost-Time (CT): The cost-time value for a problem instance solved to optimality (i.e., achieved a cost value of zero) is less than one. Table II and Fig. 4 show the probability distribution of the mean cost-time and the average for each algorithm. Instances with CT $<$ 2 are those which were frequently solved by a given algorithm.

B. Predicting Performance

Using the success rate as the performance metric, we considered two absolute thresholds for good performance: SR $>$ 0 and SR $>$ 0.5. We also considered two absolute thresholds for mean cost-time value: CT $<$ 2 and CT $<$ 2.5.

In the feature selection algorithm (SIFTED), the number of features is specified by the user. For each of the thresholds, we ran MATILDA multiple times with a different number of features to evaluate how this affects the selected features, and the quality of the trained SVMs. To determine the number of features for each threshold we considered the average model accuracy, precision and recall from 10-fold cross-validation. We also considered the F1 score, which is the harmonic mean of precision and recall. Based on these metrics, the models with 10 features were selected for all thresholds, except SR $>$ 0.5, for which we used 6 features.

Each subfigure in Fig. 5 visualises performance with respect to a performance metric. The bars are grouped by whether good performance is specified in terms of SR or CT, and the choice of threshold. The first four bars correspond to the performance of the SVM that predicts whether each algorithm will achieve good performance on a problem instance. The last bar, coloured purple, depicts the performance of the model trained to select the best performing algorithm out of the four considered based on the rules outlined in Section III-D. This model is called the Selector.

When SR is used to define good performance, higher thresholds result in increased class imbalance between good (class 1) and bad (class 0) performance. Because of the small proportion of class 1 instances, the classes are easily separable. Consequently, the SVMs trained on SR $>$ 0.5 have near perfect accuracy and recall. When good performance is defined as SR $>$ 0, the trained SVMs still perform reasonably well, but the error rate is higher.

The choice of threshold is also important when good performance is defined in terms of CT. When the threshold is 2, which is closer to the average CT for SA and RR, the SVMs trained to predict whether SA and RR will achieve lower performance compared to the corresponding SVMs for

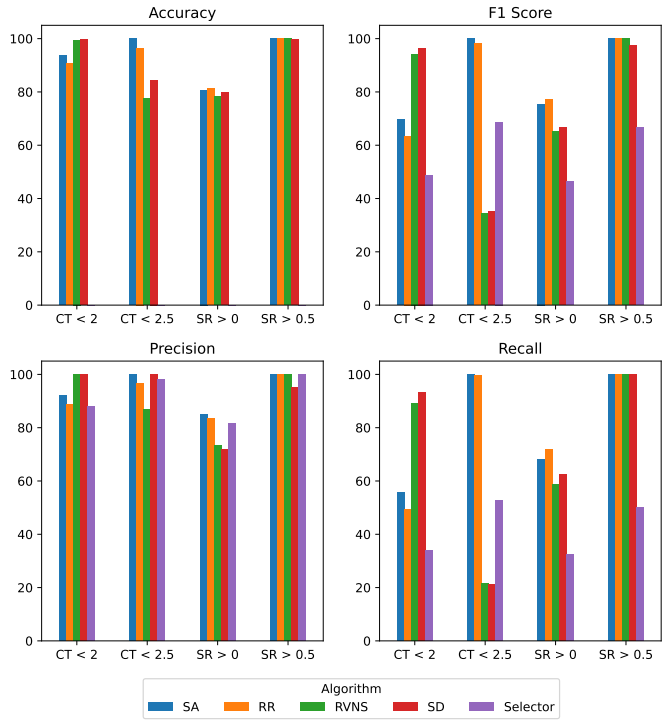

Fig. 5. Average model evaluation metrics as percentages for SVMs trained to predict good performance and models to select best algorithm by MATILDA

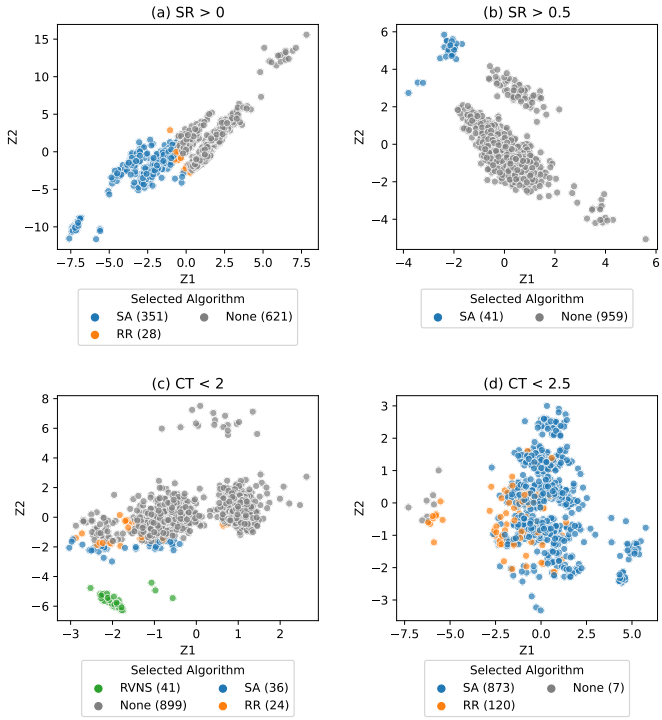

Fig. 6. Projection of instance space showing the best algorithm and the number of instances for which an algorithm is selected as best

RVNS and SD. When the CT threshold is 2.5, which is closer to the average for SD and RVNS, the SVMs for these methods perform worse than those for SA and RR.

Fig. 6 shows which algorithm is selected by the Selector as the best for each instance in the projected instance space. When SR $>$ 0 and SR $>$ 0.5 are used to define good performance, if none of the algorithms exhibit good performance, SA is recommended as it has the best average SR performance. Overall when assessing performance in terms of SR, SA is generally the superior algorithm. According to MATILDA when good performance is defined in terms of CT $<$ 2 or CT $<$ 2.5, RR is recommended in cases where none of the algorithms perform well. This may seem counter-intuitive, particularly in the case of the CT $<$ 2.5 Selector, where SA is selected in the vasts majority of instances. On all of the 873 instances for which SA is selected, RR also performs well. Selecting either of the algorithms would lead to similar results. However, based on the MATILDA selection rules, SA is chosen for these instances because its SVM has higher precision compared to that of the RR SVM (as seen in Fig. 5).

C. Understanding Performance

Table III summarises the features found to best characterise the problem instances in terms of algorithm performance. Fig. 7 shows that there are similar trends between the success rates and the projected instance space for all four algorithms. Based on these projections, as well as the correlation of the features with each other, the algorithms appear to have greater success solving puzzles with the following characteristics: a higher fraction of integer variables in the solution of the LP relaxation (`LP_fracInt`), more naked singles and pairs (`counts_naked1`, `counts_naked2`), greater average graph clustering coefficient of the GCP reformulation (`GCP_clustcoef`) and shorter paths for all possible vertex pairs in the graph of the GCP reformulation (`GCP_avg`).

Similarly, the algorithms find solutions closer to a cost of zero and more quickly for instances with the following characteristics: a higher fraction of integer variables in the solution of the LP relaxation (`LP_fracInt`), more naked pairs (`counts_naked2`), denser graphs from the GCP reformulation (`GCP_density`) and a lower mean number of empty cells that can potentially take each value (`value_mean`). However correlation between these features and the mean cost-time is not as strong for RVNS and SD as it is for the other algorithms.

We can also explain the algorithm selection for a given instance, particularly for the models using mean cost-time as a performance metric, by considering Fig. 6 and the projection matrices. For example, in plot (c), although RR is recommended for the vast majority of the instances (including those that "None" is selected), there is a cluster of instances at the lower left corner of the instance space that RVNS is actually the best algorithm. These instances have a high count of naked singles and pairs and a low variation coefficient for the variable integer slack of the LP relaxation.

TABLE III
FEATURES SELECTED BY SIFTED ALGORITHM

Feature	Description	SR > 0	SR > 0.5	CT < 2	CT < 2.5
counts_CV	CV of the count of possible values each empty cell can take given the fixed cells		*		
counts_min	minimum count of possible values each empty cell can take given the fixed cells	*		*	*
counts_naked1	number of empty cells that can take only 1 possible value given the fixed cells	*	*	*	*
counts_naked2	number of empty cells that can take only 2 possible value given the fixed cells	*		*	*
counts_naked3	number of empty cells that can take only 3 possible value given the fixed cells	*			
fixedDig_max	maximum number of times each value appears as a fixed cell				*
value_max	maximum number of empty cells that can potentially take each value			*	*
value_mean	mean number of empty cells that can potentially take each value			*	*
value_min	minimum number of empty cells that can potentially take each value			*	*
GCP_avgPath	average length of the shortest paths for all possible vertex pairs in graph of GCP formulation	*			
GCP_clustcoef	average graph clustering coefficient of the GCP formulation	*			
GCP_density	density of the graph of GCP formulation			*	*
GCP_nDeg_std	standard deviation of vertex degrees of GCP formulation			*	
LP_fracInt	fraction of variables set to 0 or 1 in solution of LP relaxation of SAT formulation	*	*	*	*
LPslack_CV	variable integer slack statistics of LP relaxation of SAT formulation			*	*
LPslack_entropy	variable integer slack statistics of LP relaxation of SAT formulation	*	*		
SAT_ratioLin	the linearised clause-to-variable ratio of the SAT formulation	*			
SAT_ratioRec	reciprocal of the clause-to-variable ratio of the SAT formulation	*	*		
VG_CV	node degree statistics for the variable graph of the SAT formulation		*		

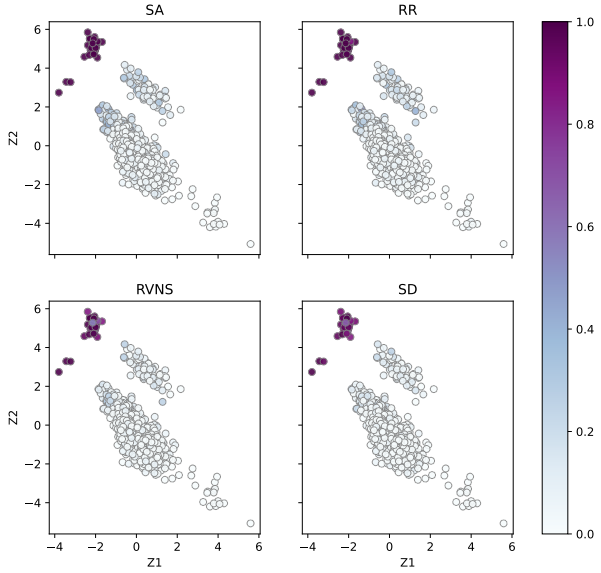

Fig. 7. Instance space showing the success rate for each algorithm using the PILOT projection from the SR > 0.5 models

D. Comparison with Logistic Regression

We next compare MATILDA to multinomial logistic regression models with l_1-penalty for feature selection to select the best algorithm. These models were all solved using SAGA [26], implemented in the *scikit-learn* library in Python [27]. For each of the regression models, the inverse of the regularisation strength (C) was tuned using 5-fold cross-validation. The greater this value, the more features had non-zero coefficients and were included in the model. Five classes were defined: one for each of the algorithms and "None". Instances are given the label "None" if three or more algorithms tie for best performance. This is the case for 406 instances when success rate is used as the performance metric.

When training a model based on success rate, ten features had non-zero coefficients. Several of these were the same as those selected by MATILDA. In contrast to MATILDA, none of the SAT reformulation features were included, while some of the fixed cell distribution variables (`fixedDig_min`, `fixedDig_entropy`) were selected. The multinomial logistic regression model selected SA for 338 instances, RR for 4 instance and predicted that for 658 of the instances, there will be no difference in the performance of the algorithms. Table IV shows that this model has relatively poor accuracy and precision (estimated through 10-fold cross validation). So while the selections are rather similar to those made by MATILDA, MATILDA also has the benefit of better predicting good performance to support decision making.

When training a logistic regression model based on mean cost-time, C was selected to be very small, and no features were included in the model. Instead RR is selected for all instances. Interestingly, this naive classifier results in better selections than those made by MATILDA.

TABLE IV
AVERAGE MODEL EVALUATION METRICS FOR MULTINOMIAL LOGISTIC REGRESSION MODELS

	Success Rate	Mean Cost-Time
Accuracy (%)	50.7	78.3
Precision (%)	62.9	83.0
Recall (%)	50.7	78.3
F1 score (%)	37.8	68.8

V. CONCLUSION

In this paper we studied the instance space analysis and algorithm selection problems for solving Sudoku puzzles using metaheuristics. We considered a large number of problem instances, which is atypical for the application of metaheuristics to Sudoku [2], [8]. We also constructed a large number of features to characterise puzzles and identified features that are informative of algorithm performance. Our findings suggest that not only the logic-related features are useful, but also features derived from reformulations. In particular, the GCP

reformulation provides useful information pertaining to the performance of metaheuristic solvers.

Our analysis focused on the Melbourne Algorithm Test Instance Library with Data Analytics (MATILDA). An important limitation of MATILDA is that the classification performance of the final predictive models is highly dependent on how the classes are defined with respect to the original (non categorical) performance measures [7]. When success rate was used to distinguish between "good" and "bad" performance, SA was selected as the best solver for nearly all instances. When mean cost-time was used, the selection was more nuanced, and the final predictive models produced by MATILDA did not manage to outperform a naive prediction. Therefore, the additional complexity of MATILDA proved worthwhile when using success rates, but not when using mean cost-time as a performance metric.

When the performance of (some of the) metaheuristic solvers is similar, predicting whether an algorithm will perform "well" is more meaningful compared to trying to identify the best performing solver. This also suggests that a more appropriate evaluation metric should take into account the cost of an incorrect prediction (an aspect that is ignored by typical measures of performance). This constitutes a direction for future research. A limitation of our work is that it focused exclusively on puzzles of the same order. In future work, puzzles of higher order (16×16 or 25×25 grids) can be considered to understand how the algorithms used scale with size. We may also include logic-based solvers in the algorithm set and compare them to the metaheuristic solvers.

ACKNOWLEDGMENTS

This paper is based on work completed while Danielle Notice was part of the EPSRC funded STOR-i Centre for Doctoral Training (EP/S022252/1). This work was also funded in part by Tesco Stores Limited.

REFERENCES

[1] R. Lewis, "Metaheuristics can solve sudoku puzzles," *Journal of Heuristics*, vol. 13, no. 4, pp. 387–401, 2007.

[2] R. H. Chae and A. C. Regan, "An analysis of harmony search for solving Sudoku puzzles," *Soft Computing Letters*, vol. 3, p. 100017, 2021.

[3] C. Blum and A. Roli, "Metaheuristics in combinatorial optimization: Overview and conceptual comparison," *ACM Computing Surveys*, vol. 35, no. 3, pp. 268–308, 2003.

[4] D. H. Wolpert and W. G. Macready, "No free lunch theorems for optimization," *IEEE Transactions on Evolutionary Computation*, vol. 1, no. 1, pp. 67–82, 1997.

[5] J. R. Rice, "The algorithm selection problem," in *Advances in Computers*, M. Rubinoff and M. C. Yovits, Eds. Elsevier, 1976, vol. 15, pp. 65–118.

[6] K. Smith-Miles and L. Lopes, "Measuring instance difficulty for combinatorial optimization problems," *Computers & Operations Research*, vol. 39, no. 5, pp. 875–889, 2012.

[7] K. Smith-Miles and M. A. Muñoz, "Instance space analysis for algorithm testing: methodology and software tools," *ACM Computing Surveys*, 2022.

[8] A. Z. Sevkli and K. A. Hamza, "General variable neighborhood search for solving sudoku puzzles: unfiltered and filtered models," *Soft Computing*, vol. 23, no. 15, pp. 6585–6601, 2018.

[9] K. Smith-Miles, M. Muñoz, and Neelofar, "Melbourne algorithm test instance library with data analytics (MATILDA)," 2020. [Online]. Available: https://matilda.unimelb.edu.au/

[10] T. Dillon, "Tdoku: A fast Sudoku solver and generator," 2019, (Accessed: June 1, 2022). [Online]. Available: https://github.com/t-dillon/tdoku

[11] S. Kirkpatrick, C. D. Gelatt Jr, and M. P. Vecchi, "Optimization by simulated annealing," *Science*, vol. 220, no. 4598, pp. 671–680, 1983.

[12] G. Dueck, "New optimization heuristics," *Journal of Computational Physics*, vol. 104, no. 1, pp. 86–92, 1993.

[13] P. Hansen, N. Mladenović, and J. A. Moreno Pérez, "Variable neighbourhood search: methods and applications," *Annals of Operations Research*, vol. 175, no. 1, pp. 367–407, 2009.

[14] T. Mantere and J. Koljonen, "Sudoku research page," (Accessed: November 1, 2022). [Online]. Available: http://lipas.uwasa.fi/ timan/sudoku/

[15] J. Rosenhouse and L. Taalman, *Taking Sudoku Seriously : The Math Behind the World's Most Popular Pencil Puzzle*. Oxford University Press, 2012.

[16] "Sudoku of the day," (Accessed: December 1, 2022). [Online]. Available: https://www.sudokuoftheday.com/difficulty

[17] T. Davis, "The mathematics of Sudoku," 2012, unpublished. [Online]. Available: http://www.geometer.org/mathcircles/sudoku.pdf

[18] L. C. Coelho and G. Laporte, "A comparison of several enumerative algorithms for Sudoku," *Journal of the Operational Research Society*, vol. 65, no. 10, pp. 1602–1610, 2014.

[19] R. Lewis, "Applications and extensions," in *A Guide to Graph Colouring*. Springer Cham, 2016, ch. 5, pp. 111–149.

[20] K. Smith-Miles, D. Baatar, B. Wreford, and R. Lewis, "Towards objective measures of algorithm performance across instance space," *Computers & Operations Research*, vol. 45, pp. 12–24, 2014.

[21] E. Nudelman, K. Leyton-Brown, H. H. Hoos, A. Devkar, and Y. Shoham, "Understanding random SAT: beyond the clauses-to-variables ratio," in *Principles and Practice of Constraint Programming*, M. Wallace, Ed. Springer Berlin Heidelberg, 2004, pp. 438–452.

[22] I. Lynce and J. Ouaknine, "Sudoku as a SAT problem," in *9th International Symposium on Artificial Intelligence and Mathematics*, 2006.

[23] M. Muñoz and K. Smith-Miles, "Instance space analysis: A toolkit for the assessment of algorithmic power," 2021 2020.

[24] M. A. Muñoz, L. Villanova, D. Baatar, and K. Smith-Miles, "Instance spaces for machine learning classification," *Machine Learning*, vol. 107, no. 1, pp. 109–147, 2017.

[25] D. Notice, "ASP Sudoku," 2023, (Accessed: April 24, 2023). [Online]. Available: https://github.com/danotice/ASP-Sudoku

[26] A. Defazio, F. Bach, and S. Lacoste-Julien, "SAGA: A fast incremental gradient method with support for non-strongly convex composite objectives," *Advances in Neural Information Processing Systems*, vol. 27, 2014.

[27] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.