European Journal of Operational Research 315 (2024) 484–498

Contents lists available at ScienceDirect

European Journal of Operational Research

journal homepage: www.elsevier.com/locate/eor

Discrete optimization

Instance space analysis for 2D bin packing mathematical models

Chang Liu [a,b,c,*], Kate Smith-Miles [a,c], Tony Wauters [b], Alysson M. Costa [a,c]

[a] School of Mathematics and Statistics, The University of Melbourne, Parkville, Melbourne, Australia
[b] Department of Computer Science, CODeS, KU Leuven, Gebroeders De Smetstraat 1, 9000 Ghent, Belgium
[c] ARC Training Centre in Optimisation Technologies, Integrated Methodologies, and Applications (OPTIMA), The University of Melbourne, Parkville, Melbourne, Australia

ARTICLE INFO

Keywords:
Packing
Instance space analysis
Instance generation
Performance evaluation
Mixed integer programming

ABSTRACT

In this paper, we apply Instance Space Analysis (ISA) to study the two-dimensional bin-packing problem. We consider classical and newly-generated instances to test the performance of four mixed-integer programming (MIP) models from the literature. This is the first time ISA is used to compare MIP models. We set as a performance metric the time taken by the black-box MIP solver CPLEX to obtain a proven optimal solution when running each model. Our results provide a new perspective on the different models' performance according to each instance's features.

1. Introduction

It is infamously difficult to assess algorithm performance objectively (Hooker, 1994). The standard practice of reporting algorithms' 'on average' performance across a suite of test instances can lead to misleading conclusions if the instances' diversity and suitability are not scrutinized. An algorithm that performs well on a given or chosen set of problem instances is not inherently guaranteed to do well with different instances (Hooker, 1995). The No Free Lunch Theorem (Wolpert & Macready, 1997) suggests that an algorithm that performs better for some instances should perform worse for others. Merely considering average performance does not provide insight into an algorithm's strengths and weaknesses.

This does not mean we must abandon the practice of testing and comparing algorithm performance on benchmark instances. Instead, we should establish whether the benchmark instances have the properties that allow broad conclusions to be drawn; the instances should be demonstrably diverse, lacking bias, sufficiently challenging, discriminating of algorithms' performances, and representative of real-world applications (Smith-Miles & Geng, 2022). If the benchmarks lack these properties, conclusions drawn about algorithm performances will have limited validity and applicability. To resolve this, new instances with the required characteristics should be generated (Hall & Posner, 2010).

The Instance Space Analysis (ISA) methodology, recently developed by Smith-Miles with others (Smith-Miles & Muñoz, 2023) , provides a robust framework for analyzing algorithms' performances. The methodology projects problem instances onto a 2D space known as the *instance space*, with each instance corresponding to a point on the instance space. This projection is made by calculating features of instances that explain algorithm performance and determining a mapping from these features to the instance space.

Mapping to a 2D space allows for easier visualization of the problem instances under consideration. We can identify regions where a given algorithm is strong or weak by overlaying algorithm performance metrics across the instance space. The visual presentation of instances also allows us to scrutinize the diversity and unbiasedness of our initial set of problem instances. If our instances lack diversity, we can attempt to generate more instances in desired regions of the space. We can apply machine learning techniques to this experimental data to learn the region of the instance space where each algorithm is expected to obtain good performance. This region is called the *algorithm footprint*. These footprints can be combined to form an algorithm selector, which selects the algorithm most expected to perform well for any given instance. Ultimately, this provides far more insight into the structure of the problem than merely reporting the average performances of algorithms.

Instance Space Analysis has been applied to many problems, including combinatorial optimization problems. Smith-Miles and Bowly (2015) and Smith-Miles et al. (2014) are seminal papers in the development of Instance Space Analysis, and they both applied their techniques to the graph coloring problem. The former developed the methodology behind Instance Space Analysis, and the latter demonstrated evolving instances to fill gaps in the instance space. More recently, ISA has been applied to the 0-1 knapsack problem, beginning with Smith-Miles et al. (2021), which developed features for the

* Corresponding author at: School of Mathematics and Statistics, The University of Melbourne, Parkville, Melbourne, Australia.
E-mail addresses: cliu2@student.unimelb.edu.au (C. Liu), smith-miles@unimelb.edu.au (K. Smith-Miles), tony.wauters@kuleuven.be (T. Wauters), alysson.costa@unimelb.edu.au (A.M. Costa).

https://doi.org/10.1016/j.ejor.2023.12.008
Received 14 April 2023; Accepted 7 December 2023
Available online 12 December 2023
0377-2217/© 2023 The Authors. Published by Elsevier B.V. This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/4.0/).

problem and identified hard instances. This work was extended in Jooken et al. (2022, 2023), which developed computationally expensive features that provide further insight into what makes a knapsack problem instance hard.

Another application in optimization is the maximum flow problem (Alipour et al., 2023). This recent paper expanded the ISA methodology by developing an instance selection procedure known as filtering. By removing similar instances from the metadata in a controlled manner, this procedure improves the uniformity of the density of instances across the space while retaining important information from the instances. Instance Space Analysis has also been applied to continuous black-box optimization problems. Muñoz and Smith-Miles (2020) considers the single objective problem, while Yap et al. (2022) expands the analysis to multi-objective optimization. Both papers demonstrate how new instances with controllable properties can be generated.

Other applications to optimization problems include workforce scheduling (Kletzander et al., 2021), job shop scheduling (Strassl & Musliu, 2022), and inventory routing (Skålnes et al., 2023). ISA has also been applied to the curriculum-based course timetabling problem (De Coster et al., 2022; Smith-Miles & Muñoz, 2023).

Instance Space Analysis has also been used to examine machine learning and related problems, starting with classification problems. Muñoz et al. (2018) considered each classification dataset as an instance and developed features of datasets to explain classifier performance. In contrast, Paiva et al. (2022) adapted the ISA framework to analyze individual classification datasets, with each observation being an instance. This adapted framework is presented in the open-source Python package PyHard, which includes an interactive visualization interface relating instance hardness to classifier performance. ISA has also been applied to regression problems (Muñoz et al., 2021), clustering (Fernandes et al., 2021), and facial age estimation (Smith-Miles & Geng, 2022). It can be applied to any problem where algorithms are developed and evaluated on test instances, and the test instances can be described with features that explain their difficulty.

In this paper, Instance Space Analysis is applied to mixed-integer programming (MIP) models for the first time. MIP formulations of the 2D bin packing problem are treated as our 'algorithms.' Their performance will be assessed by whether a MIP solver (in this case, CPLEX 22.1.1 with default settings) can solve the models within a given time limit. It is important to note that the performance of a MIP model depends on the solver used, and some models may be better suited to one solver than another. In Appendix A, we confirm that a different choice of solver (Gurobi 10.0.1) results in only small differences in the models' solvability.

The two-dimensional bin packing problem (2BP) is a combinatorial optimization problem that involves packing a set of smaller rectangles (items) into a minimal number of larger rectangles (bins). A feasible solution to a 2BP instance consists of a placement of all items in bins without overlap. A variant of the 2BP imposes the guillotine constraint, which mandates that the items in each bin can be obtained by applying a series of edge-to-edge cuts. However, in this paper, we do not enforce the guillotine constraint.

The rest of the paper is structured as follows. Section 2 will present the MIP models that are analyzed, Section 3 will explain the Instance Space Analysis methodology in more detail, and will present the metadata for the initial instance space, the results of which will be shown in Section 4. Section 5 will discuss the process of generating new instances to fill gaps in the initial space, and Section 6 will present the results of the updated instance space. Finally, Section 7 will present the conclusions drawn from this paper and directions for future research.

2. Mixed-integer linear programming models

In this paper, we compare different mixed-integer programming (MIP) models for the 2BP. The first MIP model for a multi-dimensional bin packing problem was presented in Chen et al. (1995). The model was originally developed for the 3D container loading problem but can be adapted to the two-dimensional case. Other MIP models for the 2BP have been proposed in the papers (Blum et al., 2012; Castro & Oliveira, 2011; Pisinger & Sigurd, 2007). In addition to the models, Pisinger and Sigurd (2007) presented a branch-price-and-cut approach for solving the 2BP, while Blum et al. (2012) proposed a heuristic that combines a constructive one-pass heuristic with an evolutionary algorithm. Both Blum et al. (2012)and Pisinger and Sigurd (2007) presented their models for pedagogical reasons and were not necessarily focusing on fine-tuning their characteristics to enhance the performance of MIP solvers. Nevertheless, the models have interesting design choices and they are worth exploring. Castro and Oliveira (2011) focused on developing a model for the 2D strip packing problem, a variant of the 2BP where we have a single strip of fixed width and unlimited height, and we want to fit all the items into the strip with minimal height. However, the authors also discuss how the model can be modified for the 2BP.

An instance of the 2BP can be described by the following data:

- W and H: the width and height of the bins,
- n: the number of items,
- w_i and $h_i, \forall i \in \{1, \dots, n\}$, the widths and heights of the items.

We assume that all widths and heights are positive integers and that $W = H$. Note that the latter choice does not reduce the generality of the instances as individual items' widths and heights can be scaled to represent situations where $W \neq H$. We also calculate possible upper and lower bounds (UB, LB, respectively) for each instance. The upper bound is obtained by running a state-of-the-art heuristic: the goal-driven ruin and recreate approach from Gardeyn and Wauters (2022). While the heuristic is for the 2BP with guillotine constraints, the solution obtained will still be feasible for the 2BP without guillotine constraints. The number of bins used by the solution gives a feasible UB. For the lower bound, we use the continuous lower bound $LB = \left\lceil \frac{\sum_{i=1}^{n} w_i h_i}{WH} \right\rceil$.

Both the lower and upper bounds are used when formulating MIPs. The lower bound is applied similarly for each model, with a constraint stating that the objective is greater than or equal to the lower bound. The upper bound is used to define the scope of the problem variables.

In the presentation of the MIP models below, we will refer to each model by the first author of the paper in which it was presented. When writing the models, some variables have been renamed from their original papers to highlight similarities between different models. Additionally, some modifications were made to the Chen and Castro models to give them a more level playing field. Specifically, Constraints (9) and (10) for the Chen model were added since other models have similar optional 'symmetry-breaking' constraints. For the Castro model, we apply the principle of normal patterns (Christofides & Whitlock, 1977). This decreases the number of potential positions for items that need to be considered, reducing the number of variables needed to define the model.

2.1. Chen's model

The Chen model was developed for the container loading problem (Chen et al., 1995), but it can easily be adapted to the 2BP. The paper proposes additional constraints relevant to container loading problems, for example, weight distribution constraints. These constraints do not apply to the 2BP and are omitted below.

$$\text{min:} \quad \sum_{k=1}^{UB} q_k, \tag{1}$$

$$\text{s.t.:} \quad \sum_{i=1}^{n} s_{i,k} \leq n q_k, \qquad \forall k, \tag{2}$$

$$\sum_{k=1}^{UB} s_{i,k} = 1, \qquad \forall i, \tag{3}$$

$$x_i + w_i \le W, \qquad \forall i, \tag{4}$$
$$y_i + h_i \le H, \qquad \forall i, \tag{5}$$
$$x_i + w_i \le x_j + W(1 - l_{i,j}), \qquad \forall i, j, i \ne j, \tag{6}$$
$$y_i + h_i \le y_j + H(1 - u_{i,j}), \qquad \forall i, j, i \ne j, \tag{7}$$
$$s_{i,k} + s_{j,k} - 1 \le l_{i,j} + l_{j,i} + u_{i,j} + u_{j,i}, \qquad \forall i, j, k, i < j, \tag{8}$$
$$q_{k-1} \le q_k, \qquad \forall k, k > 1, \tag{9}$$
$$\sum_{k=1}^{i} s_{i,k} = 1, \qquad \forall i, i < UB, \tag{10}$$
$$LB \le \sum_{k=1}^{UB} q_k, \tag{11}$$
$$q_k \in \{0,1\}, \qquad \forall k, \tag{12}$$
$$s_{i,k} \in \{0,1\}, \qquad \forall i, k, \tag{13}$$
$$l_{i,j}, u_{i,j} \in \{0,1\}, \qquad \forall i, j, i < j, \tag{14}$$
$$x_i, y_i \in \mathbb{Z}_{\ge 0}, \qquad \forall i. \tag{15}$$

Chen Model Formulation

The indexes for all variables and constraints are: $i, j \in \{1, \dots, n\}$ indexing over items and $k \in \{1, \dots, UB\}$ indexing over bins. The variables have the following meanings:

- q_k is a binary variable that is 1 if bin k is used,
- $s_{i,k}$ is a binary variable that is 1 if item i is in bin k,
- $l_{i,j}, u_{i,j}$ are binary variables equal to 1 if item i is placed to the left or under item j, respectively. They are defined for $i \ne j$,
- (x_i, y_i) are the coordinates of item i's bottom-left corner.

Constraints (2) ensure that bins can only contain items if used, and Constraints (3) guarantee that every item is placed in a bin. Conditions (4) and (5) ensure that every item lies inside their bin.

Constraints (6) and (7) ensure that items do not overlap. This non-overlapping check is only needed if the items are in the same bin, as imposed by inequalities (8). Constraints (9) and (10) help reduce the problem's symmetry. The former ensures that a bin can only be used if all of the lower index bins are used, while the latter ensures that each item is placed in a bin with an index no greater than the item's index. For example, item 1 must be placed in bin 1, and item 2 must be placed in bin 1 or 2. Finally, constraint (11) applies the lower bound.

2.2. Pisinger's model

The Pisinger model was presented in Pisinger and Sigurd (2007). The paper briefly discusses the model, mentioning that the model is difficult to solve using ordinary MIP solvers.

min: z (16)

s.t.:
$$m_i \le z, \qquad \forall i, \tag{17}$$
$$m_i \le UB, \qquad \forall i, \tag{18}$$
$$x_i + w_i \le W, \qquad \forall i, \tag{19}$$
$$y_i + h_i \le H, \qquad \forall i, \tag{20}$$
$$x_i + w_i \le x_j + W(1 - l_{i,j}), \qquad \forall i, j, i \ne j, \tag{21}$$
$$y_i + h_i \le y_j + H(1 - u_{i,j}), \qquad \forall i, j, i \ne j, \tag{22}$$
$$m_i + 1 \le m_j + UB(1 - p_{i,j}), \qquad \forall i, j, i \ne j, \tag{23}$$
$$l_{i,j} + l_{j,i} + u_{i,j} + u_{j,i} + p_{i,j} + p_{j,i} \ge 1, \qquad \forall i, j, i < j, \tag{24}$$
$$m_i \le i, \qquad \forall i, \tag{25}$$
$$LB \le z, \tag{26}$$
$$l_{i,j}, u_{i,j} \in \{0,1\}, \qquad \forall i, j, i < j, \tag{27}$$
$$p_{i,j} \in \{0,1\}, \qquad \forall i, j, i \ne j, \tag{28}$$
$$x_i, y_i \in \mathbb{Z}_{\ge 0}, \qquad \forall i, \tag{29}$$
$$m_i \in \mathbb{Z}_{>0}, \qquad \forall i, \tag{30}$$
$$z \in \mathbb{Z}_{>0}. \tag{31}$$

Pisinger Model Formulation

The main difference between this model and Chen's is the variables responsible for assigning items to bins. In the Pisinger model, we have an integer variable for each item equal to the index of the bin to which it is assigned, while in the Chen model, we have binary variables for each (item, bin) pair. Let $i, j \in \{1, \dots, n\}$ be the indexes for items for all variables and constraints. As in the Chen model:

- $l_{i,j}, u_{i,j}$ are binary variables that are 1 if item i is placed to the left or under item j, respectively. They are defined for $i \ne j$,
- (x_i, y_i) are the coordinates of item i's bottom-left corner.

Moreover, the following variables are defined:

- z is an integer variable indicating the number of bins used,
- m_i is an integer variable equal to the index of the bin that contains item i,
- $p_{i,j}$ is a binary variable equal to 1 if item i is in an earlier (lower index) bin than item j.

Inequalities (19)–(22) for Pisinger's model are the same as inequalities (4)–(7) for Chen's model. Constraints (25) and (26) have the same purpose as Constraints (10) and (11), respectively; the first constraints ensure that each item is placed in a bin with an index no greater than the item's index and the second constraint applies the lower bound.

Conditions (17) guarantee that the objective corresponds to the highest bin index used. Constraints (18) apply the upper bound to the bin indexes. Constraints (23) and (24), like Constraints (8) for Chen's model, ensure that items must be non-overlapping if they are in the same bin. Constraints (23) ensure that if items i and j are in the same bin, $p_{i,j} = p_{j,i} = 0$. In that case, inequalities (24) force one of the non-overlapping variables to be equal to 1, which ensures that the items do not overlap.

2.3. Blum's model

The Blum model was presented in the paper Blum et al. (2012), which was primarily focused on developing a heuristic for the 2BP. The paper does not discuss the difficulty of solving this model.

min: $\sum_{i=1}^{n} s_{i,i}$ (32)

s.t.:
$$s_{i,k} \le s_{k,k}, \qquad \forall i, k, i \ge k, \tag{33}$$
$$\sum_{k=1}^{i} s_{i,k} = 1, \qquad \forall i, \tag{34}$$
$$x_i + w_i \le W, \qquad \forall i, \tag{35}$$
$$y_i + h_i \le H, \qquad \forall i, \tag{36}$$
$$x_i + w_i \le x_j + W(3 - l_{i,j} - s_{i,k} - s_{j,k}), \qquad \forall i, j, k, i \ne j, k \le min(i, j), \tag{37}$$
$$y_i + h_i \le y_j + H(3 - u_{i,j} - s_{i,k} - s_{j,k}), \qquad \forall i, j, k, i \ne j, k \le min(i, j), \tag{38}$$
$$l_{i,j} + l_{j,i} + u_{i,j} + u_{j,i} = 1, \qquad \forall i, j, i < j, \tag{39}$$
$$\sum_{i=1}^{n} s_{i,i} \le UB, \tag{40}$$
$$LB \le \sum_{i=1}^{n} s_{i,i}, \tag{41}$$

$$l_{i,j}, u_{i,j} \in \{0,1\}, \qquad \forall i,j,i<j, \tag{42}$$

$$s_{i,k} \in \{0,1\}, \qquad \forall i,k,i \geq k, \tag{43}$$

$$x_i, y_i \in \mathbb{Z}_{\geq 0}, \qquad \forall i. \tag{44}$$

Blum Model Formulation

Let $i,j \in \{1,\dots,n\}$ index over items and $k \in \{1,\dots,n\}$ index over bins for all variables and constraints. Note that, for the Blum model, items and bins have the same indexing set. As in the Chen model:

- $l_{i,j}, u_{i,j}$ are binary variables that are 1 if item i is placed to the left or under item j, respectively. They are defined for $i \neq j$,
- (x_i, y_i) are the coordinates of item i's bottom-left corner.

Moreover, the following variables are defined:

- $s_{i,k}$ is a binary variable that is 1 if item i is in bin k. It is defined only for $i \geq k$.

In the Blum model, we initialize n bins, each corresponding to an item with the same index. A bin is considered open only if the corresponding item is placed in that bin. Additionally, every item must be placed in its corresponding or an earlier bin. For example, item 1 can only be placed in bin 1. Item 3 can be placed in bin 3, bin 1, which is always open since item 1 must be placed there, or bin 2 only if item 2 is also placed in bin 2. Despite these restrictions, all combinations of items packed into a bin are still possible, but they must go in the bin corresponding to the lowest index item in the combination.

Constraints (34), like Constraints (3) for Chen's model, ensure that every item is placed in a bin, and conditions (35) and (36) are the same as conditions (4) and (5). Constraints (37)–(39) are slightly different from Constraints (6)–(8) for Chen's model, but both sets of constraints ensure that items do not overlap. Inequality (41), like Chen's inequality (11), applies the lower bound.

Constraints (33), like Constraints (2), ensure that items are only placed in open bins. For the Blum model, bins are open if they contain the item with the same index. Inequality (40) applies the upper bound.

Compared to Chen's model, Blum's assignment of items to bins attempts to reduce the problem's symmetry, possibly making it easier for MIP solvers. The two models use a different number of binary variables for assigning items in bins. The Blum model uses $\frac{n(n+1)}{2}$ binary variables for placing items in bins, while the Chen model uses $n \cdot UB$ binary variables. Chen's model uses fewer binary variables if UB is small, while Blum's model uses fewer variables if UB is large.

2.4. Castro's model

The Castro model was presented in Castro and Oliveira (2011). This is a very different model from the previous ones. It is a discrete formulation consisting entirely of binary variables. A binary variable is defined for each necessary combination of {item, bin, x-coordinate, y-coordinate}, yielding $n \cdot UB \cdot W \cdot H$ binary variables. The principle of normal patterns (Christofides & Whitlock, 1977) outlined below is applied to reduce the number of binary variables.

Let $i \in \{1,\dots,n\}$ be the index for items and $k \in \{1,\dots,UB\}$ be the index for bins. We can formally define the set of normal positions as $X = \{\sum_{i=1}^{n} w_i \epsilon_i | \epsilon_i \in \{0,1\}\}$ and $Y = \{\sum_{i=1}^{n} h_i \epsilon_i | \epsilon_i \in \{0,1\}\}$. We want to use these normal positions to model the possible positions of the items, so we can further define $X' = \{x \in X | x \leq W - min(w_i)\}$ and $Y' = \{y \in Y | y \leq H - min(h_i)\}$ since every item must lie inside the bin. Finally, for each item i, we define $X_i = \{x \in X | x \leq W - w_i\}$ and $Y_i = \{y \in Y | y \leq H - h_i\}$ since each item must lie inside its bin.

$$\text{min:} \quad \sum_{k=1}^{UB} q_k, \tag{45}$$

$$\text{s.t.:} \quad r_{k,x,y} = q_k - \sum_{i=1}^{n} \sum_{\substack{x' \in X_i \\ x - w_i + 1 \leq x' \leq x}} \sum_{\substack{y' \in Y_i \\ y - h_i + 1 \leq y' \leq y}} n_{i,k,x',y'}, \qquad \forall k, x \in X', y \in Y', \tag{46}$$

$$\sum_{x \in X_i} \sum_{y \in Y_i} \sum_{k=1}^{UB} n_{i,k,x,y} = 1, \qquad \forall i, \tag{47}$$

$$q_{k-1} \leq q_k, \qquad \forall k, k > 1, \tag{48}$$

$$\sum_{x \in X_i} \sum_{y \in Y_i} \sum_{k=1}^{i} n_{i,k,x,y} = 1, \qquad \forall i, i < UB, \tag{49}$$

$$LB \leq \sum_{k=1}^{UB} q_k, \tag{50}$$

$$q_k \in \{0,1\}, \qquad \forall k, \tag{51}$$

$$r_{k,x,y} \in \{0,1\}, \qquad \forall k, x \in X', y \in Y', \tag{52}$$

$$n_{i,k,x,y} \in \{0,1\}, \qquad \forall i,k, x \in X_i, y \in Y_i. \tag{53}$$

Castro Model Formulation

As in the Chen model:

- q_k is a binary variable equal to 1 if bin k is used.

Moreover, the following variables are defined:

- $r_{k,x,y}$ is a binary variable equal to 1 if the unit square with bottom-left corner at (x,y) in bin k is unoccupied by any item, and 0 if a single item occupies it. It is defined for $x \in X', y \in Y'$,
- $n_{i,k,x,y}$ is a binary variable equal to 1 if item i is placed in bin k with bottom-left coordinate (x,y). These are only defined for $x \in X_i, y \in Y_i$.

Constraints (46) are equivalent to the combination of Constraints (2) and (4)–(8). If bin k is unused, then Constraints (46), like Constraints (2), state that no items are placed in the bin. If bin k is used, it ensures that at most one item occupies each square (x,y) in the bin. Like Constraints (4)–(8), this ensures that items do not overlap.

Constraints (47), like Constraints (3) for the Chen model, ensure that every item is placed in some position in a bin. Constraints (48) and (49) are symmetry-breaking constraints; Constraints (48), like Constraints (9) in the Chen model, ensure that a higher index bin can only be used if the lower index bins are also used. Constraints (49), like Constraints (10), guarantee that each item is placed in a bin with an index no greater than the item's index. Finally, inequality (50), like inequality (11), imposes the lower bound.

It is sufficient to consider only the normal positions as the possible positions of the items. Any arrangement of items in a bin can be turned into an equivalent *normal pattern*, a pattern in which all items have bottom-left coordinates in the set $X \times Y$. Intuitively, this can be done by repeatedly moving every item to its left and/or down until its edge touches that of another item or the bin until no further movement is possible for any item. Since we assume every item is in a normal position, Constraints (46) only need to check if the 1×1 squares corresponding to these normal positions are occupied.

2.5. Experimental settings

All MIP models were implemented and run using CPLEX 22.1.1 with a single thread, default settings, and a time limit of 3600 s. All experiments were performed on the Spartan HPC system (Meade et al., 2017), with an Intel(R) Xeon(R) Gold 6254 CPU @ 3.10 GHz processor and 4 GB memory.

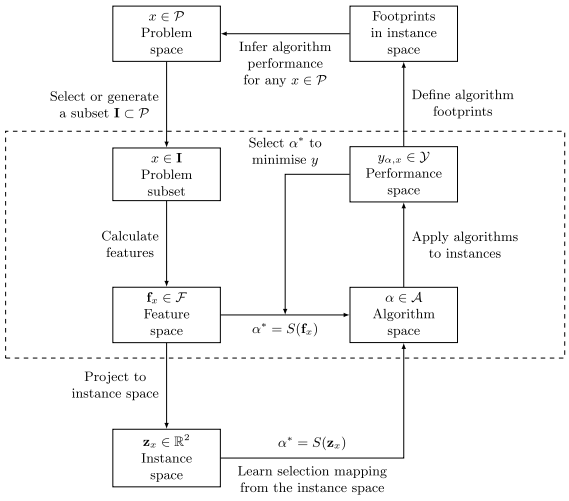


Fig. 1. Instance Space methodology framework in Smith-Miles et al. (2014), extending the original Algorithm selection problem framework from Rice (1976), shown in the dotted box.

2.6. Comparison of models

In this section, we have presented four MIP models for the 2BP. It is natural to ask what properties of instances could lead to one model being easier to solve than another. Treating Chen's model as the baseline, Pisinger's model is very similar, except that it uses a handful of integer variables for assigning items to bins rather than a larger number of binary variables. Both the range of the integer variables for the Pisinger model and the number of binary variables for the Chen model will increase with the upper bound UB. For larger values of UB, the two models will be more different; the Pisinger model will have larger ranges for its integer variables, and the Chen model will use more binary variables.

In contrast, Blum's model also uses binary variables to assign items to bins. It uses more variables than Chen's model for smaller values of UB and fewer for larger values of UB. Finally, Castro's model is substantially different from all other models. Since the number of variables is proportional to the square of the bin length, we would expect Castro's model to be challenging to solve for instances with a greater bin length. Additionally, Castro would likely be easier to solve for instances where normal patterns significantly reduce the number of variables and constraints.

These observations suggest that the models' solvability would depend on the instances' characteristics. We should be able to develop features that explain model solvability. We can then perform an Instance Space Analysis to compare the models' performances. Each model's strengths and weaknesses can be identified in the instance space, and we can identify the features that best distinguish these regions. Ultimately, this will provide insight into both the structure of the 2BP and the MIP models.

3. Instance space analysis: Methodology and meta-data

Instance Space Analysis is a methodology developed by Smith-Miles and others (Smith-Miles et al., 2014; Smith-Miles & Bowly, 2015; Smith-Miles & Lopes, 2012; Smith-Miles & Muñoz, 2023) which extends the original Algorithm Selection Problem framework proposed in Rice (1976). Fig. 1 illustrates the framework of Instance Space Analysis and its component spaces.

In the top left is the ill-defined *problem space* $\mathcal{P}$, which includes all instances of the 2BP. Since there are infinitely many problem instances, obtaining computational results for the entire problem space is impossible. Instead, Instance Space Analysis is always performed on a subset $\mathbf{I} \subset \mathcal{P}$ of the problem space. For each instance, we calculate characteristics of the instance called *features*, determining a mapping to *feature space*. These features should be able to explain the performances of our algorithms. The *algorithm space* $\mathcal{A}$ consists of the portfolio of algorithms that are being analyzed. The *performance space* $\mathcal{Y}$ consists of a performance metric that can be computed for each (algorithm, instance) pair.

The collection $\{\mathbf{I}, \mathcal{F}, \mathcal{A}, \mathcal{Y}\}$ is called the *meta-data*. Instance Space Analysis is performed on this meta-data to obtain a mapping g from feature space to *instance space*. By presenting every instance as a point in 2D space, we can visually identify each algorithm's *footprint*, the region of the instance space where each algorithm is predicted to perform well. From the instance space, we can learn a selection mapping S that chooses the algorithm α^* most likely to perform well for any given instance. We can identify the features that best explain algorithm performance and instance difficulty by observing trends in the selected features.

Instance Space Analysis is an iterative process. After the initial instance space is created, we identify gaps in the instance space and try to fill those gaps with new instances, which are added to our problem subset $\mathbf{I}$. The entire procedure of computing the meta-data and performing Instance Space Analysis is repeated until the instance space contains no gaps and the features explaining algorithm performance have stabilized. The following sections present the meta-data for applying Instance Space Analysis to the 2BP.

Table 1
Description of CLASS 01-06 instances.

Class	Bin size	Item sizes
1	10 × 10	$[1, 10] \times [1, 10]$
2	30 × 30	$[1, 10] \times [1, 10]$
3	40 × 40	$[1, 35] \times [1, 35]$
4	100 × 100	$[1, 35] \times [1, 35]$
5	100 × 100	$[1, 100] \times [1, 100]$
6	300 × 300	$[1, 100] \times [1, 100]$

3.1. Instance subset

In the initial Instance Space Analysis, we consider 420 test instances: 100 from the literature (Berkey & Wang, 1987; Martello & Vigo, 1998) and 320 obtained from a generator (Berkey & Wang, 1987). In this analysis, we will only consider instances with 40 items so that the number of items does not become the single dominant feature.

The CLASS instances are the most widely used 2BP instances for algorithm testing. These are ten classes of instances; CLASS01-CLASS06 were proposed in Berkey and Wang (1987), and CLASS07-CLASS10 were proposed in Martello and Vigo (1998). Each class consists of 10 instances for each of $n = 20, 40, 60, 80, 100$. This creates a set of 100 instances (10 instances x 10 classes) with 40 items each, which we denote as the CLASS instances.

In classes 1–6, the bins have a fixed side length, and the widths and heights of the items are randomly and uniformly generated integers from a given interval. The bin sizes and item size intervals are shown in Table 1.

Instances from classes 7–10 have bin lengths of 100, and each instance consists of four types of items. Each of those four types has widths and heights that are randomly and uniformly generated from the following intervals:

- Type 1: width generated from $[\frac{2}{3}W, W]$, height generated from $[1, \frac{1}{2}H]$
- Type 2: width generated from $[1, \frac{1}{2}W]$, height generated from $[\frac{2}{3}H, H]$

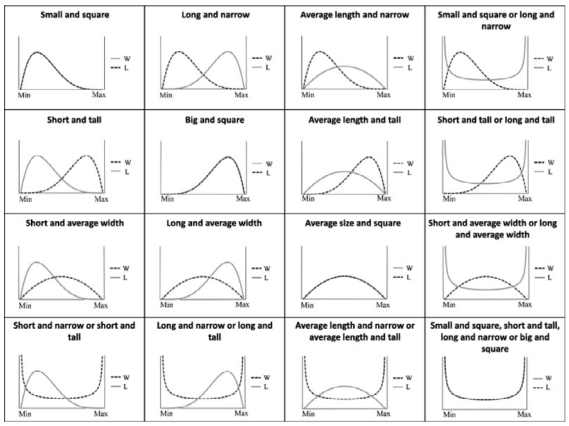


Fig. 2. Options for the sizes and shapes of the items for GEN instances. *Source:* Figure from Silva et al. (2014)

- Type 3: width generated from $[\frac{1}{2}W, W]$, height generated from $[\frac{1}{2}H, H]$
- Type 4: width generated from $[1, \frac{1}{2}W]$, height generated from $[1, \frac{1}{2}H]$

Each of the four classes has 70% of their items from one type and 10% from the others. For example, CLASS07 has 70% of its items from Type 1 and CLASS08 has 70% of its items from Type 2. These instances were developed to model more real-world situations.

To enhance the diversity and quantity of test instances, we generate additional instances using the 2DCPackGen generator (Silva et al., 2014) with slight modifications. These instances are generated with $n = 40$ and the side length of the bins chosen randomly and uniformly from the interval $[10, 110]$. The items are generated from 16 different distributions. For each distribution ('item type'), 20 instances are generated, corresponding to 320 generated instances. We denote this set of 320 instances as the GEN instances. Each distribution is denoted by its item type number; for example, GEN01. The distributions are shown in Fig. 2. Altogether, we have 100 CLASS instances and 320 GEN instances for 420 in total.

3.2. Feature space

For a successful and insightful instance space analysis, we need to develop and measure features that explain instance difficulty and algorithm performance. We can draw inspiration for features from analyzing instances that are trivial to solve. For example, an instance that only has items with width and height greater than half the bin length would be straightforward to solve since every item could only fit in a separate bin. An instance that only contains extremely tiny items, so all items easily fit in a single bin, would also be trivial to solve. This motivates features like Huge, which measures the proportion of items that have an area greater than half the area of the bin, and Tiny, which measures the proportion of items with an area less than one-tenth of the bin's area.

Another example of relatively easy instances consists of items with a fixed width w and varying heights. Since the widths are constant, solving these instances would merely involve solving the much easier 1D bin packing problem for the heights. The solution for the 1D problem can be adapted to the 2BP by fitting the 1D configurations side by side as columns with width w in the 2D bins. Features Varwidth and Varheight measure the variances in the widths and heights of the items, respectively.

A RemainingArea feature close to 1 ensures that any optimal solution will have a lot of unused area. Hence, optimal solutions are likely to be more flexible and easier for the solver to find. For example, if the total area of the items is 1.1 times the area of a bin, the RemainingArea feature would be 0.9. An optimal solution will use at least 2 bins, so at least 0.9 times the area of a bin will be unused.

Table 2
Table of features.

No.	Feature	Description
1	W	The side length of the bins.
2	Mean	The mean of the areas of the items.
3	Var	The variance of the areas of the items.
4	Skewness	The skewness of the areas of the items.
5	Kurtosis	The kurtosis of the areas of the items.
6	Huge	The proportion of items that have area $> \frac{1}{2}$.
7	Large	The proportion of items that have area $> \frac{1}{3}$ but $\leq \frac{1}{2}$.
8	Small	The proportion of items that have area $> \frac{1}{10}$ but $\leq \frac{1}{4}$.
9	Tiny	The proportion of items that have area $\leq \frac{1}{10}$.
10	Meanwidth	The mean of the widths of the items.
11	Varwidth	The variance of the widths of the items.
12	Skewnesswidth	The skewness of the widths of the items.
13	Kurtosiswidth	The kurtosis of the widths of the items.
14	Meanheight	The mean of the heights of the items.
15	Varheight	The variance of the heights of the items.
16	Skewnessheight	The skewness of the heights of the items.
17	Kurtosisheight	The kurtosis of the heights of the items.
18	MeanLSratio	The mean of the LSratios of the items.
19	VarLSratio	The variance of the LSratios of the items.
20	SkewnessLSratio	The skewness of the LSratios of the items.
21	KurtosisLSratio	The kurtosis of the LSratios of the items.
22	Square	The proportion of items with LSratio $\leq \frac{5}{4}$.
23	Uneven	The proportion of items with LSratio ≥ 4.
24	RemainingArea	The gap between the total area of the items and it rounded up to the nearest integer, $\left\lceil \sum_{i=1}^{n} \frac{w_i h_i}{WH} \right\rceil - \sum_{i=1}^{n} \frac{w_i h_i}{WH}$.

Table 2 presents the 24 features computed for each instance. When calculating the features, all widths, heights, and areas of items are considered relative to the size of the bins. For example, when computing the Meanwidth feature, we find the mean of w_i/W, rather than the mean of w_i. The area of an item is the product of its width and height, and the longer-to-shorter ratio (LSratio) is the ratio between the item's longer and shorter side lengths. The LSratio equals 1 for perfectly square items; for all other items, it is larger. All features are easy to calculate and do not require sorting the items.

Most of the features are standard statistical measures for the areas (2–5), widths (10–13), heights (14–17), and LSratios (18–21) of the items. Feature 1 measures the bin's size, which impacts the number of possible positions for each item. Features 6–9 provide additional information about the sizes of the items, allowing us to distinguish different classes of instances with varying item areas.

Features 22 and 23 measure the proportion of almost square and highly rectangular items, respectively. These features help capture the two-dimensional nature of the 2BP rather than just measuring widths, heights, or areas in isolation.

3.3. Algorithm space

The 'algorithms' used for the instance space analysis are four MIP models, Chen, Pisinger, Blum, and Castro, described in Section 2. This is the first time that instance space analysis has been applied to assess the performance of models rather than algorithms. However, the methodology is still applicable once we define a performance measure for the models. In Sections 4–6, 'model' and 'algorithm' will be used

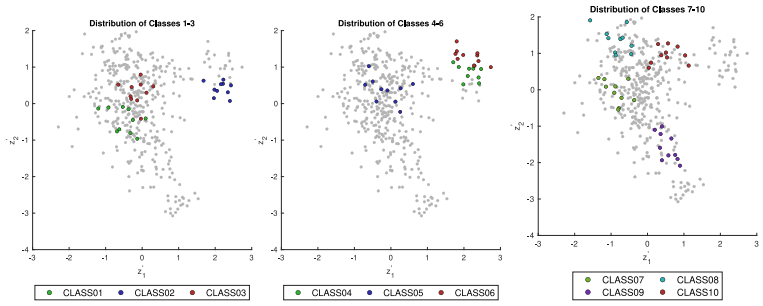


Fig. 3. Distribution of CLASS instances in the initial instance space.

interchangeably. When referring to a model, we often shorten our description to just the first author's name. For example, we may shorten 'the Chen model' to simply 'Chen.'

3.4. Performance space

Our performance measure is the wall-clock time (in seconds) that CPLEX 22.1.1 takes to solve a model to proven optimality. If an instance cannot be solved within the time limit of 3600 s, we arbitrarily set the measure to 3600.

We consider a model to be 'good' for a given instance if the instance can be solved to proven optimality within the given time limit. Equivalently, good performance means the performance measure is strictly less than 3600. This metric means that there will be instances where every model is good or every model is bad.

Alternative definitions of good can be used. For example, we could define a model as good if it is best, i.e., the fastest, or within some acceptable range from the best. In this paper, we are more interested in whether a model can be solved with the time limit than how long it takes, so we will not use these other definitions of good performance. If model A takes 1 s to solve, and model B takes 5 min, we still want to recognize that model B could be solved rather than deeming it bad since it took longer to solve than model A. This choice is further justified by the fact that wall-clock time can be noisy, especially for quickly solved models, and it can vary significantly when experiments are repeated.

Other performance measures can be used to assess the performance of solvers on MIP models. For example, we can consider the relative MIP gap, which is the relative difference between the best upper and lower bounds found by the solver. This performance measure can be used to develop a different definition of good performance, such as considering it good if the relative gap is less than 5%. In this paper, we are most interested in whether the MIP model is solved to optimality. Our definition of good performance is straightforward and avoids the need for a (possibly arbitrary) choice of conversion gap.

4. Initial instance space

4.1. Feature selection

We will provide an outline of the steps involved in Instance Space Analysis below. For full details, we refer the reader to the recent paper Smith-Miles and Muñoz (2023).

To create the instance space, we must determine the projection from feature space to instance space. Firstly, we apply the PRELIM (Preparation for Learning of Instance Meta-data) method. This prepares the meta-data by calculating a binary measurement of good performance for each instance and algorithm pair based on our definition of good performance. PRELIM also pre-processes the raw features by applying a Box–Cox transformation followed by a Z transformation. This tries to ensure that every feature is distributed like a standard normal distribution.

Feature selection is then obtained via the SIFTED (Selection of Instance Features to Explain Difficulty) routine, consisting of two steps. In the first step, features that are insufficiently correlated with the performance of any algorithm are removed, ensuring that only features that can potentially explain algorithm performance remain. Specifically, a feature is removed if the correlation between the feature and every algorithm's binary performance is lower than 0.2. Secondly, correlation clustering analysis is applied to the remaining features to ensure that a variety of features are selected. If several features measure the same properties of the instances and thus are highly correlated, they will be in the same cluster and only one will be chosen. We limit our analysis to 10 clusters to ensure that only 10 features are selected.

After applying both steps, the PILOT (Projecting Instances with Linearly Observable Trends) routine is applied to obtain an optimal projection from the 10-dimensional feature space to the 2D instance space. PILOT tries to ensure algorithm performance and feature values vary as linearly as possible throughout the instance space. The features selected and the projection obtained are shown in equation (54).

$$\begin{bmatrix} Z_1 \\ Z_2 \end{bmatrix} = \begin{bmatrix} 0.0512 & 0.1672 \\ -0.5103 & 0.2272 \\ 0.2632 & 0.3902 \\ -0.2273 & -0.2504 \\ -0.0592 & 0.2450 \\ -0.1608 & 0.0989 \\ -0.0289 & 0.0543 \\ 0.2458 & -0.2685 \\ -0.1313 & 0.1763 \\ 0.2110 & 0.0069 \end{bmatrix}^T \begin{bmatrix} W \\ Small \\ Tiny \\ Meanwidth \\ Skewnesswidth \\ Varheight \\ SkewnessLSratio \\ Square \\ Uneven \\ RemainingArea \end{bmatrix} \tag{54}$$

After determining the projection using PILOT, we run the CLOISTER (Correlated Limits of the Instance Space's Theoretical or Experimental Regions) routine to estimate the theoretical boundary of the instance space. CLOISTER uses the features' empirical upper and lower bounds and the correlations between the features to estimate the boundary of the instance space. Specifically, we consider the set consisting of all vertices corresponding to a combination of the empirical upper and lower bounds for each selected feature. These vertices form a bounding box that encloses all instances in our problem subset; however, some of the vertices represent combinations that would be unlikely to occur. To remedy this, we rejected vertices by comparing the correlations of features, using 0.7 as the threshold. If two features are positively correlated with a correlation greater than the threshold, we remove vertices where one feature takes its lower bound value and the other takes its upper bound value, since this combination of features is unlikely to coincide. If they are negatively correlated with a correlation below −0.7, we remove vertices containing both upper or both lower bounds. The remaining vertices are projected to instance space using Eq. (1), and the estimated boundary is defined as the boundary of these points' convex hull.

The position of the CLASS instances, color-coded by class number, in the instance space is shown in Fig. 3, while the location of the GEN instances is shown in Fig. 4. The holistic distribution of CLASS and

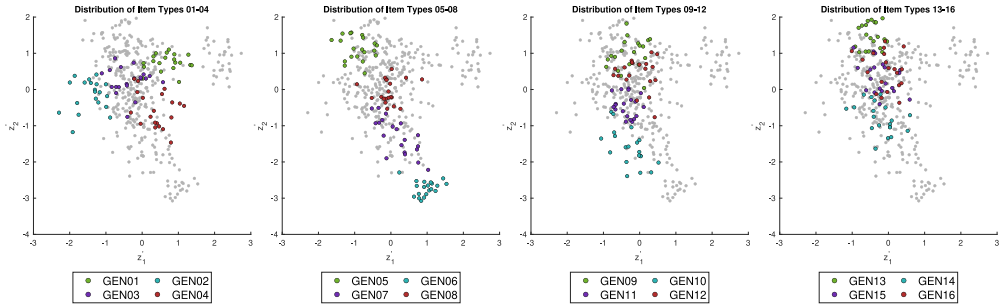


Fig. 4. Distribution of GEN instances in the initial instance space.

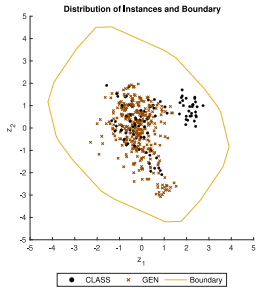


Fig. 5. Distribution of instances and boundary in the initial instance space.

Table 3
Average performance of 2BP models for initial instances.

Source	Chen	Pisinger	Blum	Castro	Total
CLASS	59 (59.0%)	40 (40.0 %)	47 (47.0 %)	19 (19.0 %)	100
GEN	144 (45.0 %)	42 (13.1 %)	98 (30.6 %)	138 (43.1%)	320
Total	203 (48.3 %)	82 (19.5 %)	145 (34.5 %)	157 (37.4 %)	420

GEN instances and the estimated boundary of the instance space are shown in Fig. 5. There is a substantial gap between the boundary and the CLASS and GEN instances, indicating that there may be instances with very different properties to our current benchmarks that are not included in the current analysis. Section 5 will investigate the possibility of generating instances to fill these gaps.

4.2. Algorithm performance and feature distribution

Recall that a model is considered 'good' if it can be solved within the time limit. Table 3 displays each model's number of good performances for CLASS and GEN instances. The table shows that Chen is the best-performing model on average, being able to solve around 48% of all instances. It is the best algorithm for both CLASS and GEN instances. Pisinger is the worst-performing model overall, as it can only solve less than 20% of the instances. Blum and Castro perform around equally well on average. Blum is a bit better for CLASS instances, while Castro is the worst model for CLASS instances but is better for GEN instances.

While Chen is the best model on average, it may have weaknesses not shown in the table. There may be instances that Chen cannot solve, but other models can. On the other hand, Pisinger is the worst algorithm overall, but there is no guarantee that it is equally bad for all instances; there may be instances with specific characteristics for which Pisinger performs equally or better than other models. To analyze and explain strengths and weaknesses, the sources of the instances are not as important as their properties. We can see from Figs. 3–5 that instances from different sources can lie in the same part of the instance space, indicating that the instances' properties are similar. For example, CLASS09 and GEN07 instances overlap around $(0.5, -1.5)$. Even though the instances come from different sources, they have similar properties and lie in similar regions of the instance space.

As a result, it is insufficient to merely consider average performances across sources, and we should look at the models' performances throughout instance space. Fig. 6 displays the (binary) performance distributions of the models in the instance space. Blue dots represent good performance, and orange dots represent bad performance. The plots in Fig. 7 provide a visual picture of how the important features vary across the instance space. (Plots of all selected features are included in the Supplementary Material, available in the online version.) By analyzing these figures, we can identify trends affecting algorithm performance.

Chen is the strongest algorithm overall, as indicated in Table 3. It can solve almost all instances at the bottom and right sides of the instance space and some in the middle. It cannot solve instances on the left side of the space, for example, near $(-1, 0.5)$, but most of the other algorithms also struggle in that region.

Instances at the bottom of the instance space have significant Square and Meanwidth features. This means they have large widths, and since the items are close to square, they will also have large heights. The bottommost instances near $(1, -3)$ are from the generator with item type 6. Fig. 2 shows that GEN06 corresponds to big, square items. These and other instances near them, including CLASS09, GEN07, GEN10, and GEN14, present a particular challenge to the Pisinger model. This is Pisinger's distinctive region of weakness since it cannot solve instances in this area, while the other models can. Note that while these instances are from different sources, they have a similar profile in terms of algorithm performance. This reinforces the idea that the properties of the instances are more important than their source. All of these instances consist of many large items.

Blum solves fewer instances than Chen overall. Their difference is particularly notable in the top-middle regions near $(-0.5, 2)$ and $(0, 1)$. In this region, the Skewnesswidth feature is distinctive. A significant Skewnesswidth feature usually corresponds to a right tail in the distribution of item widths. If item heights are distributed similarly, most items will have a small area, but if an item falls in the right tail of both distributions, it will have an outlier large area. This makes sense when looking at the sources in this region of the instance space. CLASS 10 has 70% 'type 4' items, which are relatively small, but the other 30% of their items, especially the 10% of items that are type 3, will have larger areas and may appear as outliers. From the generator, item type 1, which has instances in this region, has primarily small items, but since most items will be tiny, the few larger items will have outlier areas.

Pisinger is significantly worse than Chen at the bottom of the instance space. However, it can solve instances at the top-right and top-middle with $Z_1 > 0$ and $Z_2 > 1$. The instances in the top-right region near $(2, 1)$ are all CLASS instances, specifically, classes 02, 04, and 06. The feature distributions in Fig. 7 show that this region can be identified by the Tiny feature, meaning that they have items much smaller than their bins. All models except for Castro find this an easy region of the instance space, with only a handful of instances

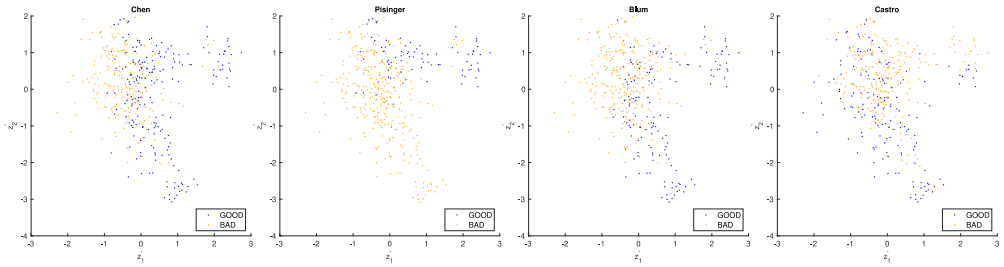


Fig. 6. Performance distribution of models in the initial instance space.

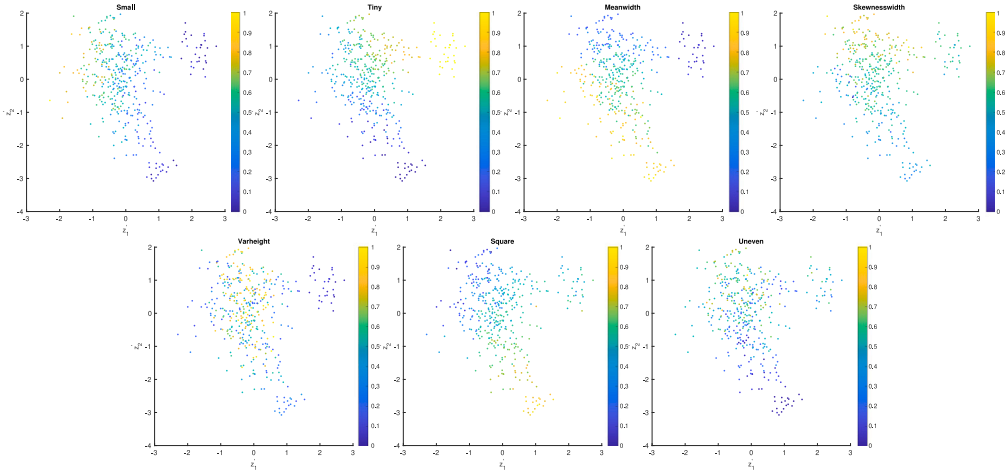


Fig. 7. Distribution of important features in the initial instance space.

that cannot be solved within the time limit. Castro can only solve the handful of instances at the bottom near $(2, 0.5)$, corresponding to CLASS02 items.

While Chen is generally the best algorithm, Castro can perform better towards the left side of the instance space near $Z_1 = -1$. Small is the most distinguishing feature in this region of the instance space, indicating a higher proportion of small but not tiny items. The Meanwidth feature is also prominent in Castro's footprint, suggesting that Castro performs better when item widths are larger. Castro performs better for instances with larger items and worse for instances with tiny items.

The most challenging instances are in the middle and middle-left of the instance space, near $(0, 0)$ and $(-1, 1)$. In this part of the instance space, no model consistently performs well. The Uneven and Varheight features are relatively large in this region, suggesting that these instances have items with various widths and heights, including items that are long in one dimension and short in the other.

4.3. Algorithm footprints and algorithm selection

After obtaining the projection from the feature space and creating the instance space, the next step is to build the performance prediction models for each algorithm. We run the PYTHIA (performance prediction and automated algorithm selection) method, using the LIBSVM library (Chang & Lin, 2011) to train a Support Vector Machine (SVM) using polynomial kernels for every algorithm. The SVM takes the instance space coordinates as inputs and outputs a binary performance measure indicating if the algorithm is expected to be good or bad. PYTHIA self-tunes the SVM parameters for each algorithm, using 30 iterations of the random search algorithm with points from Latin hypercube sampling and model error as the loss function. A 10-fold cross-validation experiment is used to evaluate the SVMs. Fig. 8 shows the SVM performance predictions for all models.

These SVM predictors are combined to form an SVM selector. For every point on the instance space, the selector chooses the model most trusted to perform well. The selector will choose the SVM with the highest precision if multiple models are predicted to perform well. If no model is expected to perform well on a point, it will choose 'None,' indicating that it has no confidence in any model. The selector results are shown in Fig. 9, and the SVM table (Table 4) shows the accuracy, precision, and recall of each algorithm's SVM and that of the selector.

The SVM predictions match our observations in Section 4.2. Chen is the algorithm with the best overall performance, Pisinger is particularly good at the top-right of the instance space, Blum is a little worse than Chen towards the middle, and Castro is a little better at the bottom-left while being much worse than the others in the top-right. The Chen model has the highest precision for any model of 87.3%, compared to the next highest of 83.7% for Blum. Hence, Chen is the model of choice in the right and bottom sides of the instance space where it is predicted to be good. Blum is only chosen in a small region near $(0, -2)$, which is consistent with our observation that it is worse than Chen overall. Castro is chosen for the instances that Chen is not expected to solve, in the bottom-left of the instance space near $(-0.5, -1.2)$. Finally, Pisinger is selected for a few instances at the top near $(0, 1.5)$, which it solves better than any other model. The 'None' region, where no algorithm is

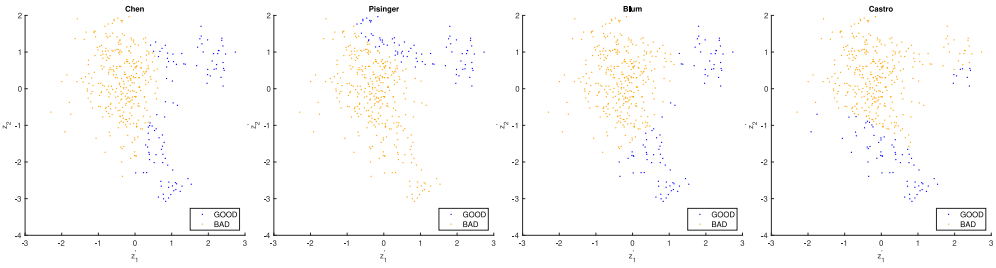


Fig. 8. Predicted performance of 2BP models in the initial instance space.

Table 4
SVM table for the initial instance space.

Algorithm	Actual percentage of good performances	Accuracy, precision, recall
Chen	48.3 %	69.8%, 87.3%, 43.8 %
Pisinger	19.5 %	90.2%, 75.9%, 73.2 %
Blum	34.5 %	80.2%, 83.7%, 53.1 %
Castro	37.4 %	72.6%, 76.9%, 38.2 %
Selector	49.5%	N/A, 78.5 %, N/A

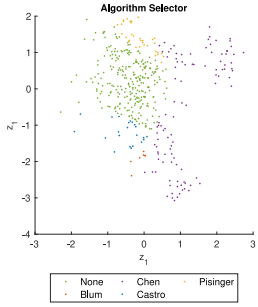


Fig. 9. Algorithm selector for the initial instance space.

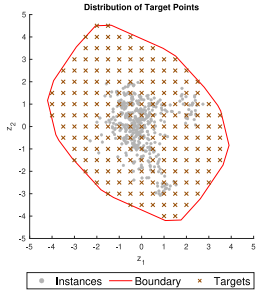


Fig. 10. Distribution of target points.

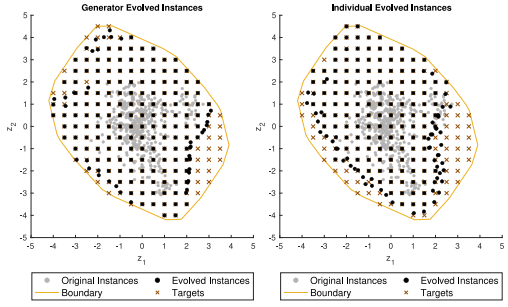


Fig. 11. Distribution of genetic algorithm evolved instances in the initial instance space.

expected to perform well, corresponds to the hard region at the middle and top-left and includes points near $(0, 0)$ and $(-1, 1)$.

5. Generating new instances using genetic algorithms

5.1. Evolving generator parameters

As shown in Fig. 5, the instances do not fill the entire region inside the estimated boundary. To enhance the diversity of the instance space, we will generate instances in areas of the instance space that were previously unoccupied. We set 192 target points throughout the instance space with the goal of generating instances that lie near these points. The target points are all the grid points separated by 0.5 that lie inside or on the boundary, and they are shown in Fig. 10.

By generating more instances and adding them to the problem subset, we can assess if the conclusions drawn in Section 4 apply to these newly generated instances. We will again keep the number of items in each instance fixed at $n = 40$.

We used two approaches to evolve instances using a genetic algorithm. The first is to evolve parameters for a generator, extending the instance generator from Silva et al. (2014). Given n, W, H, the generator from Silva et al. (2014) generates items for instances as follows: for any chosen item type between 1 and 16, the generator associates four parameters: $(\alpha_w, \beta_w, \alpha_h, \beta_h)$. For example, item type 1 corresponds to $\alpha_w = \alpha_h = 2$, $\beta_w = \beta_h = 5$, and item type 2 corresponds to $\alpha_w = \beta_h = 5$, $\beta_w = \alpha_h = 2$. For each of the beta distributions with parameters (α_w, β_w) and (α_h, β_h), we generate n values, $a_1, \ldots, a_n$ and $b_1, \ldots, b_n$ respectively, which will be real numbers in the interval $(0, 1)$. For the actual widths and heights, we scale each value to an integer between 1 and W: $w_i = \lceil W a_i \rceil$ and $h_i = \lceil W b_i \rceil$.

This framework works well for the original generator, but since there are only 16 item types, the variety of item sizes and shapes is limited. Instead of restricting their values to a discrete set, we can allow the α and β parameters to vary continuously. Using an alternative set of parameters is more convenient: (μ_w, F_w, μ_h, F_h). The μ parameters are the means of the distributions, and the F parameters affect the variances of the distributions, with the variance $F\mu(1-\mu)$ increasing linearly with F. Both parameters take values in the interval $(0, 1)$ and are easier to understand and adjust in isolation than α and β.

The (α, β) values can be obtained from the (μ, F) parameters as follows:

$$\alpha = \mu \frac{1-F}{F}, \quad \beta = (1-\mu)\frac{1-F}{F}.$$

We also want to vary the size of the bins, so our complete set of parameters for generating instances is $(W, \mu_w, F_w, \mu_h, F_h)$. W is bounded

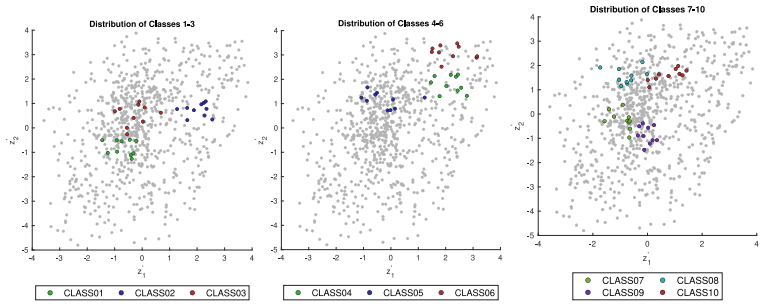


Fig. 12. Distribution of CLASS instances in the updated instance space.

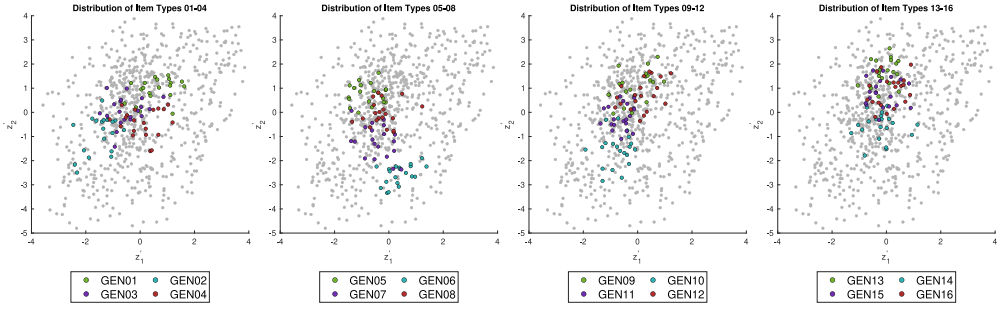


Fig. 13. Distribution of GEN instances in the updated instance space.

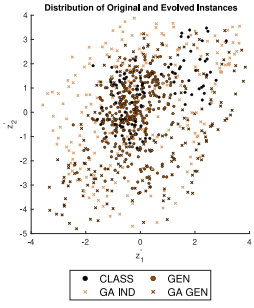


Fig. 14. Distribution of all instances in the updated instance space.

between 10 and 110 and will always be an integer. The other variables can take any real value between 0 and 1. These parameters can be varied to generate instances with many different item types. However, it is initially unclear which values of the parameters will result in instances that lie in a desired region of the instance space. We use a genetic algorithm to evolve these parameters to generate an instance as close to each target point as possible.

Once we have evolved generator parameters, we generate a single instance per target point for 192 additional instances. Each instance is generated using MATLAB's default random number generator seed to ensure replicability. While we could have saved some time and computational resources by evolving towards fewer target points and generating more instances per point, we opted for more targets and a single instance per target to increase the diversity of our new instances.

5.2. Evolving individual instances

While evolving generator parameters is effective at filling in most of the instance space, the instances can still be lacking in diversity since every item is generated from a single distribution, and the distributions of the widths and heights are always independent. Some instances cannot be generated; for example, an instance with widths and heights that vary uniformly from $[1, W]$ and $[1, H]$ but are perfectly correlated (all items are square).

An alternative approach is to evolve each instance individually by directly choosing their widths and heights. Specifically, we evolve a vector with 81 entries, consisting of the bin size, the 40 widths, and the 40 heights. The actual vector being evolved is $(W, a_1, \ldots, a_{40}, b_1, \ldots, b_{40})$. W will remain an integer between 10 and 110, but the widths and heights allowed will depend on the bin size, so we instead evolve a and b variables in the interval $(0, 1)$. To obtain the actual sizes of the items, we again scale each variable to an integer between 1 and W: $w_i = \lceil W a_i \rceil$ and $h_i = \lceil W b_i \rceil$ for all $i = 1, \ldots, 40$.

While this approach expands the pool of possible instances, the genetic algorithm will usually require more generations since we are evolving far more variables. If we want to generate many instances for each target point, evolving generator parameters would be faster since we only need to run a genetic algorithm once per target. We can then use the generator to produce as many instances as desired.

5.3. Genetic algorithm settings

For both methods, we use the same settings for the genetic algorithm, many of which are default settings of MATLAB: a genetic population size of 50, an elite count of 2, and a crossover fraction of 0.8. The initial population is randomly and uniformly generated. The genetic algorithm terminates when the average change in the fitness value is less than 10^{-6} over 50 generations. We define the fitness function for evolving generating parameters as the distance between the target point and the projection to instance space of an instance generated using the individual's values as generator parameters. The definition is similar for evolving individual instances, but here we take the difference between the target and the individual instance's projection onto the instance space.

The projections of the evolved instances onto the original instance space are shown in Fig. 11. Both approaches can generate instances throughout the instance space and significantly expand beyond the original test instances in the center.

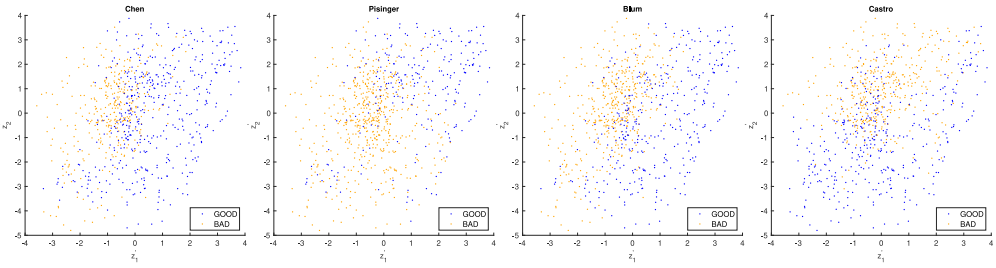


Fig. 15. Performance distribution of models in the updated instance space.

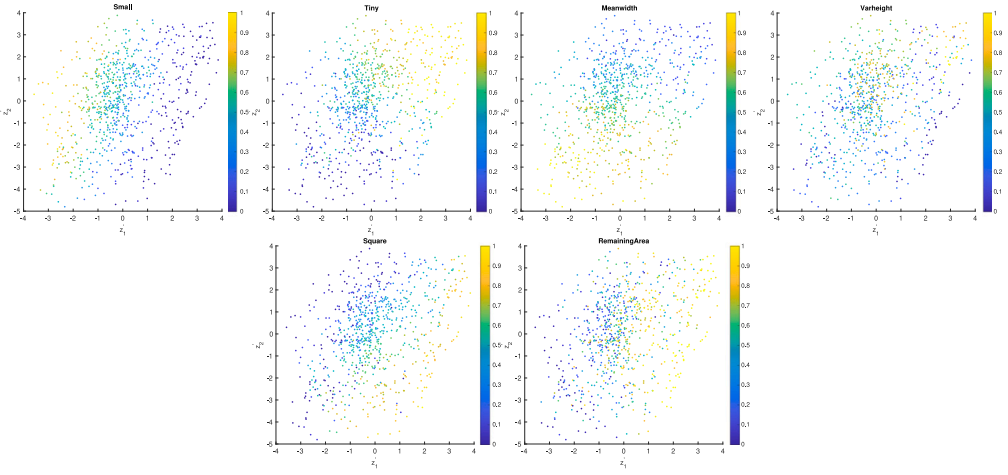


Fig. 16. Distribution of important features in the updated instance space.

6. Updated instance space

As mentioned in Section 3, Instance Space Analysis is an iterative process. Now that we have generated more instances to diversify the instance subset, we can redo the instance space analysis to check if our conclusions match those obtained from the initial instance space or if different features are chosen to explain algorithm performance. Along with the original set of 420 test instances, we have added two sets of 192 instances, one from evolving generator parameters and the other from evolving individual instances. We refer to them as GA GEN and GA IND instances, respectively. This corresponds to a total of 804 instances, shown in Table 5. The features selected and the projection obtained are shown in equation (55).

$$\begin{bmatrix} Z_1 \\ Z_2 \end{bmatrix} = \begin{bmatrix} -0.3695 & -0.3212 \\ 0.4081 & -0.4579 \\ -0.6249 & -0.2610 \\ 0.5058 & 0.3593 \\ -0.3191 & -0.3521 \\ 0.0728 & 0.1814 \\ -0.1375 & -0.3258 \\ 0.0316 & -0.3105 \\ -0.2154 & 0.4305 \\ -0.4157 & 0.1633 \end{bmatrix}^T \begin{bmatrix} W \\ Small \\ Tiny \\ Meanwidth \\ Skewnesswidth \\ Kurtosiswidth \\ Varheight \\ SkewnessLSratio \\ Square \\ RemainingArea \end{bmatrix} \tag{55}$$

To align the new instance space with the original instance space, we have applied a rotation of 210 degrees anti-clockwise about the origin, as shown in Eq. (56). These transformed coordinates will be only used for displaying and discussing the figures, and do not affect the instance space itself.

$$\begin{bmatrix} Z'_1 \\ Z'_2 \end{bmatrix} = \begin{bmatrix} cos(210°) & -sin(210°) \\ sin(210°) & cos(210°) \end{bmatrix} \begin{bmatrix} Z_1 \\ Z_2 \end{bmatrix} \tag{56}$$

The position of the CLASS instances, color-coded by class number, is shown in Fig. 12, while the location of the GEN instances, color-coded by item type, is shown in Fig. 13. The holistic distribution of CLASS, GEN, and genetically evolved instances are shown in Fig. 14.

6.1. Algorithm performance and feature distribution

Table 5 displays each model's number of good performances for CLASS, GEN, and genetically evolved instances. Fig. 15 displays the performance distributions of the models in the instance space, and Fig. 16 shows the distributions of the important features.

Table 5 suggests that GA GEN instances are the easiest instances, with every model able to solve most of them. GA IND instances are more challenging and comparable to CLASS instances in difficulty, except for the Castro model, which finds CLASS instances much more difficult. Fig. 14 shows that GA GEN instances tend to lie toward the bottom, being able to reach areas like $(z'_1, z'_2) = (-3, -4)$, while GA IND instances fill the top-left part of the instance space near $(z'_1, z'_2) = (-2, 2)$. Looking at the performance distributions in Fig. 15, instances at the top-left of the space are challenging for all models while instances at the bottom are much easier. This suggests that the features can explain the difference in average difficulty between the two sets of evolved instances.

Using the rotated coordinates, the distribution of CLASS and GEN instances are similar to the original space, with instances near the center remaining there, while instances near the outside like GEN05 and GEN06 remain at the top-left and bottom-right respectively. The features selected are almost identical to those chosen in the original instance space. The only different feature chosen is the Kurtosiswidth feature, replacing Uneven which was chosen in the original analysis.

The new instance space magnifies some trends identified in the previous analysis while revealing some new trends. The bottom and

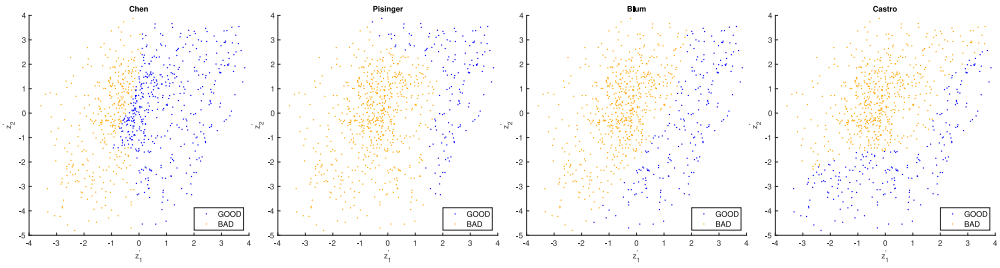


Fig. 17. Predicted performance of 2BP models in the updated instance space.

Table 5
Average performance of 2BP models for updated instances.

Source	Chen	Pisinger	Blum	Castro	Total
CLASS	59 (59.0%)	40 (40.0 %)	47 (47.0 %)	19 (19.0 %)	100
GEN	144 (45.0 %)	42 (13.1 %)	98 (30.6 %)	138 (43.1%)	320
GA GEN	143 (74.5 %)	108 (56.2 %)	115 (59.9 %)	149 (77.6 %)	192
GA IND	111 (57.8 %)	51 (26.6 %)	92 (47.9 %)	77 (40.1 %)	192
Total	457 (56.8 %)	241 (30.0 %)	352 (43.8 %)	383 (47.6 %)	804

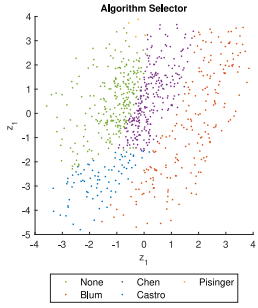


Fig. 18. Algorithm selector for the updated instance space.

top-right regions in Fig. 6 where Chen performed well were disjoint. Having added instances between those two regions, we can see that Chen also performs well in between. Blum still performs worse than Chen throughout the instance space, though it is able to solve a handful of instances at the bottom of the instance space that Chen cannot. These instances have significant Meanwidth and Square features, suggesting that Blum can solve instances with huge items.

Pisinger is a consistently strong algorithm in the top-right of the instance space, where instances have Tiny items. Still, it cannot make any inroads elsewhere, except for a handful of instances in the far bottom-left and bottom-right of the instance space near $(-3, -2)$ and $(1.5, -3.5)$ respectively. This reinforces our observation from the initial instance space that Pisinger performs very well for instances with tiny items but poorly elsewhere. Finally, Castro is strong at the instance space's bottom side, including the bottom-left region. This matches our observation from the initial instance space that Castro is good for instances with a significant Meanwidth feature and can solve some instances with a prominent Small feature even though the other algorithms struggle. Meanwhile, it finds some instances on the top side with Tiny items very difficult.

Aside from the middle region near $(0, 1)$, the new instance space has another hard region at the top-left near $(-1.5, 1.5)$. The Small feature is prominent in this region, suggesting that these hard instances consist of many items with areas between 0.1 and 0.25 times that of the bin. The Varheight feature is quite large in this region, suggesting that the items vary in size and shape. This is compatible with our observations from Section 4.2 that the hard region contains instances with various widths and heights.

Overall, the new instance space is similar to the initial instance space but expanded. The increased diversity of our instances enhances our confidence in the conclusions drawn in Section 4.

6.2. Algorithm footprints and algorithm selection

Having created the instance space, we continue to follow the procedure of instance space analysis by creating the SVM algorithm predictors and algorithm selector. The predictors are shown in Fig. 17, selector results are shown in Fig. 18, and the SVM table (Table 6) shows the accuracy, precision, and recall of each algorithm's SVM and that of the selector.

The SVM precision for all models are much higher than those for the original instance space. Blum has the highest precision of all the algorithms and is chosen throughout its predicted region of strength at the right of the instance space. Pisinger is chosen for a handful of instances at the top of the space near $(-0.25, 3.5)$. The Chen model is chosen for instances near the middle of the space, between $(0, 0)$ and $(1, 3)$ where Blum is expected to perform badly. Finally, Castro is chosen for the bottom-left of the instance space, where no other algorithm is expected to perform well. The 'None' region goes from the middle of the space to the top-left, similar to the original instance space. Overall, the selector outperforms every individual model on average throughout the instance space.

7. Conclusion

In this paper, we have applied Instance Space Analysis (ISA) to analyze mixed-integer programming (MIP) models for the 2D Bin Packing Problem. This is the first time the methodology has been applied to test the performance of MIP models. In total, 24 features were proposed to describe a comprehensive set of literature instances and used to generate algorithmic footprints for four classical models from the literature.

The analysis allowed differentiating the models' performances according to the different features generated, providing insights that were unclear with traditional 'average-based' statistics. In particular, it made clear that the Pisinger model was superior to Chen's when instances were composed of tiny items. The Blum model tended to be more efficient than the Chen model for instances requiring a large number of bins. The Castro model, in turn, showed to be somewhat complementary to the other models, being able to obtain optimal

Table 6
SVM table for the updated instance space.

Algorithm	Actual percentage of good performances	Accuracy, precision, recall
Chen	56.8 %	74.6%, 80.2%, 73.5 %
Pisinger	30.0 %	80.2%, 77.3%, 48.1 %
Blum	43.8 %	79.1%, 93.8%, 56.0 %
Castro	47.6 %	73.6%, 88.7%, 51.2 %
Selector	60.2 %	N/A, 79.5 %, N/A

Table A.7
Comparison of the number of instances solved by CPLEX and Gurobi.

Model	Both solved	CPLEX only	Gurobi only	Both unsolved	Total
Chen	429 (53.4%)	28 (3.5%)	20 (2.5%)	327 (40.7%)	804
Pisinger	225 (28.0%)	16 (2.0%)	35 (4.4%)	528 (65.7%)	804
Blum	311 (38.7%)	41 (5.1%)	36 (4.5%)	416 (51.7%)	804
Castro	379 (47.1%)	4 (0.5%)	40 (5.0%)	381 (47.4%)	804

solutions for instances that no other model could solve in the allowed time limit.

The ISA also showed that there was a lack of instances in some regions of the space. New instances were created with two different strategies: by using a genetic algorithm to tune the parameters of an instance generator from the literature and by directly evolving values for the widths and heights of individual items. The new instances allowed us to obtain a much more comprehensive instance space, which confirmed our initial observations and also provided new conclusions, such as the obtention of a region where the Blum model outperformed the others.

Our results open several avenues for future research. Firstly, new features that further differentiate each model's performance could be developed. In particular, as in Jooken et al. (2023) , focus could be placed on the development of computationally expensive features that are less helpful for online algorithm selection but can provide even greater insight into the structure of the 2BP and MIP models. Secondly, instances of related problems, such as the strip packing problem (Júnior et al., 2019), could be included in the instance space and eventually provide insight into the similarities and differences between different variants of the problems. Finally, the observation that there are regions of the instance space analysis where none of the existing models perform well could motivate the study of new models tailored to be efficient for instances with the associated features.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This work was supported by The University of Melbourne, Australia and KU Leuven, Belgium through the KU Leuven - Melbourne Joint PhD program. This research was partially funded by the Australian Government through the Australian Research Council Industrial Transformation Training Centre in Optimisation Technologies, Australia, Integrated Methodologies, and Applications (OPTIMA), Australia, Project ID IC200100009. This research was supported by The University of Melbourne's Research Computing Services and the Petascale Campus Initiative.

Appendix A. Comparison between solvers

The goal of this appendix is to answer the question: does a different choice of solver have a significant effect on the models' solvability?

To explore this question, we have implemented and attempted to solve all the models using Gurobi 10.0.1, in addition to our original results with CPLEX 22.1.1. Both solvers are given the same conditions of a single thread, default settings, and a time limit of 3600 s. Table A.7 presents, for each model, the number of instances solved by both solvers, only one solver, and neither of them. CPLEX is able to solve a greater number of Chen and Blum models, while Gurobi has an advantage with the Pisinger and Castro formulations. These differences are small relative to the total number of instances; for every model, the vast majority (over 90%) of instances are either solved by both solvers or not solved by both solvers.

Appendix B. Supplementary data

Supplementary material related to this article can be found online at https://doi.org/10.1016/j.ejor.2023.12.008.

References

Alipour, H., Muñoz, M. A., & Smith-Miles, K. (2023). Enhanced instance space analysis for the maximum flow problem. *European Journal of Operational Research*, *304*(2), 411–428.

Berkey, J. O., & Wang, P. Y. (1987). Two-dimensional finite bin-packing algorithms. *Journal of the Operational Research Society*, *38*, 423–429. http://dx.doi.org/10.1057/jors.1987.70.

Blum, C., Schmid, V., & Baumgartner, L. (2012). On solving the oriented two-dimensional bin packing problem under free guillotine cutting: Exploiting the power of probabilistic solution construction. arXiv:1209.0880.

Castro, P. M., & Oliveira, J. F. (2011). Scheduling inspired models for two-dimensional packing problems. *European Journal of Operational Research*, *215*, 45–56. http://dx.doi.org/10.1016/j.ejor.2011.06.001.

Chang, C. C., & Lin, C. J. (2011). LIBSVM. *ACM Transactions on Intelligent Systems and Technology*, *2*, 1–27. http://dx.doi.org/10.1145/1961189.1961199.

Chen, C., Lee, S., & Shen, Q. (1995). An analytical model for the container loading problem. *European Journal of Operational Research*, *80*, 68–76. http://dx.doi.org/10.1016/0377-2217(94)00002-T.

Christofides, N., & Whitlock, C. (1977). An algorithm for two-dimensional cutting problems. *Operations Research*, *25*, 30–44. http://dx.doi.org/10.1287/opre.25.1.30.

De Coster, A., Musliu, N., Schaerf, A., Schoisswohl, J., & Smith-Miles, K. (2022). Algorithm selection and instance space analysis for curriculum-based course timetabling. *Journal of Scheduling*, *25*(1), 35–58.

Fernandes, L. H. D. S., Lorena, A. C., & Smith-Miles, K. (2021). Towards understanding clustering problems and algorithms: An instance space analysis. *Algorithms*, *14*(3), 95.

Gardeyn, J., & Wauters, T. (2022). A goal-driven ruin and recreate heuristic for the 2D variable-sized bin packing problem with guillotine constraints. *European Journal of Operational Research*, *301*, 432–444. http://dx.doi.org/10.1016/j.ejor.2021.11.031.

Hall, N. G., & Posner, M. E. (2010). The generation of experimental data for computational testing in optimization. In *Experimental methods for the analysis of optimization algorithms* (pp. 73–101). Springer Berlin Heidelberg, http://dx.doi.org/10.1007/978-3-642-02538-9_4.

Hooker, J. N. (1994). Needed: An empirical science of algorithms. *Operations Research*, *42*, 201–212. http://dx.doi.org/10.1287/opre.42.2.201.

Hooker, J. N. (1995). Testing heuristics: We have it all wrong. *Journal of Heuristics*, *1*, 33–42. http://dx.doi.org/10.1007/BF02430364.

Jooken, J., Leyman, P., & De Causmaecker, P. (2022). A new class of hard problem instances for the 0–1 knapsack problem. *European Journal of Operational Research*, *301*(3), 841–854.

Jooken, J., Leyman, P., & De Causmaecker, P. (2023). Features for the 0-1 knapsack problem based on inclusionwise maximal solutions. *European Journal of Operational Research*, *311*(1), 36–55.

Júnior, A. N., Silva, E., Gomes, A. M., Soares, C., & Oliveira, J. F. (2019). Data mining based framework to assess solution quality for the rectangular 2D strip-packing problem. *Expert Systems with Applications*, *118*, 365–380. http://dx.doi.org/10.1016/j.eswa.2018.10.006.

Kletzander, L., Musliu, N., & Smith-Miles, K. (2021). Instance space analysis for a personnel scheduling problem. *Annals of Mathematics and Artificial Intelligence*, *89*(7), 617–637.

Martello, S., & Vigo, D. (1998). Exact solution of the two-dimensional finite bin packing problem. *Management Science*, *44*, 388–399. http://dx.doi.org/10.1287/mnsc.44.3.388.

Meade, B., Lafayette, L., Sauter, G., & Tosello, D. (2017). Spartan HPC-Cloud Hybrid: Delivering Performance and Flexibility. http://dx.doi.org/10.4225/49/58ead90dceaaa.

Muñoz, M. A., & Smith-Miles, K. (2020). Generating new space-filling test instances for continuous black-box optimization. *Evolutionary Computation*, *28*, 379–404. http://dx.doi.org/10.1162/evco_a_00262.

Muñoz, M. A., Villanova, L., Baatar, D., & Smith-Miles, K. (2018). Instance spaces for machine learning classification. *Machine Learning*, *107*(1), 109–147.

Muñoz, M. A., Yan, T., Leal, M. R., Smith-Miles, K., Lorena, A. C., Pappa, G. L., & Rodrigues, R. M. (2021). An instance space analysis of regression problems. *ACM Transactions on Knowledge Discovery from Data*, *15*(2), 1–25.

Paiva, P. Y. A., Moreno, C. C., Smith-Miles, K., Valeriano, M. G., & Lorena, A. C. (2022). Relating instance hardness to classification performance in a dataset: A visual approach. *Machine Learning*, *111*(8), 3085–3123.

Pisinger, D., & Sigurd, M. (2007). Using decomposition techniques and constraint programming for solving the two-dimensional bin-packing problem. *INFORMS Journal on Computing*, *19*, 36–51. http://dx.doi.org/10.1287/ijoc.1060.0181.

Rice, J. R. (1976). The algorithm selection problem. *Advances in Computers*, *15*, 65–118. http://dx.doi.org/10.1016/s0065-2458(08)60520-3.

Silva, E., Oliveira, J. F., & Wäscher, G. (2014). 2DcPackGen: A problem generator for two-dimensional rectangular cutting and packing problems. *European Journal of Operational Research*, *237*, 846–856. http://dx.doi.org/10.1016/j.ejor.2014.02.059.

Skålnes, J., Ben Ahmed, M., Hvattum, L. M., & Stålhane, M. (2023). New benchmark instances for the inventory routing problem. *European Journal of Operational Research*, S037722172300615X.

Smith-Miles, K., Baatar, D., Wreford, B., & Lewis, R. (2014). Towards objective measures of algorithm performance across instance space. *Computers & Operations Research*, *45*, 12–24. http://dx.doi.org/10.1016/j.cor.2013.11.015.

Smith-Miles, K., & Bowly, S. (2015). Generating new test instances by evolving in instance space. *Computers & Operations Research*, *63*, 102–113. http://dx.doi.org/10.1016/j.cor.2015.04.022.

Smith-Miles, K., Christiansen, J., & Muñoz, M. A. (2021). Revisiting where are the hard knapsack problems? via instance space analysis. *Computers & Operations Research*, *128*, Article 105184. http://dx.doi.org/10.1016/j.cor.2020.105184.

Smith-Miles, K., & Geng, X. (2022). Revisiting facial age estimation with new insights from instance space analysis. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, *44*, 2689–2697. http://dx.doi.org/10.1109/TPAMI.2020.3038760.

Smith-Miles, K., & Lopes, L. (2012). Measuring instance difficulty for combinatorial optimization problems. *Computers & Operations Research*, *39*, 875–889. http://dx.doi.org/10.1016/j.cor.2011.07.006.

Smith-Miles, K., & Muñoz, M. A. (2023). Instance space analysis for algorithm testing: Methodology and software tools. *ACM Computing Surveys*, *55*(12), 1–31. http://dx.doi.org/10.1145/3572895.

Strassl, S., & Musliu, N. (2022). Instance space analysis and algorithm selection for the job shop scheduling problem. *Computers & Operations Research*, *141*, Article 105661.

Wolpert, D., & Macready, W. (1997). No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation*, *1*, 67–82. http://dx.doi.org/10.1109/4235.585893.

Yap, E., Munoz, M. A., & Smith-Miles, K. (2022). Informing multiobjective optimization benchmark construction through instance space analysis. *IEEE Transactions on Evolutionary Computation*, *26*(6), 1246–1260.