Computers and Operations Research 128 (2021) 105184

Contents lists available at ScienceDirect

Computers and Operations Research

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

Revisiting *where are the hard knapsack problems?* via Instance Space Analysis

Kate Smith-Miles *, Jeffrey Christiansen, Mario Andrés Muñoz

School of Mathematics and Statistics, The University of Melbourne, Parkville, VIC 3010, Australia

ARTICLE INFO

Article history:
Received 17 June 2020
Revised 7 December 2020
Accepted 10 December 2020
Available online 18 December 2020

Keywords:
0–1 Knapsack problem
Instance Space Analysis
Instance generation
Instance difficulty
Performance evaluation
Algorithm portfolios
Algorithm selection

ABSTRACT

In 2005, David Pisinger asked the question *"where are the hard knapsack problems?"*. Noting that the classical benchmark test instances were limited in difficulty due to their selected structure, he proposed a set of new test instances for the 0–1 knapsack problem with characteristics that made them more challenging for dynamic programming and branch-and-bound algorithms. This important work highlighted the influence of diversity in test instances to draw reliable conclusions about algorithm performance. In this paper, we revisit the question in light of recent methodological advances – in the form of Instance Space Analysis – enabling the strengths and weaknesses of algorithms to be visualised and assessed across the broadest possible space of test instances. We show where the hard instances lie, and objectively assess algorithm performance across the instance space to articulate the strengths and weaknesses of algorithms. Furthermore, we propose a method to fill the instance space with diverse and challenging new test instances with controllable properties to support greater insights into algorithm selection, and drive future algorithmic innovations.

1. Introduction

It has long been recognised that rigorous evaluation of algorithm performance is a topic deserving greater attention than it typically receives (Hooker, 1995). The standard practice of reporting "on average" performance of algorithms across a given test suite, without adequately justifying that the chosen test instances are fit for purpose, risks misleading conclusions based on biased experimental results (McGeoch, 2002). This approach also offers little insight into the unique strengths and weaknesses of algorithms for particular types of test instances with different characteristics that are potentially hidden within a test suite average.

Of course, to enable fair comparisons between studies, it is important to continue the standard practice of adopting common benchmarks, inherited and shared between researchers. However, it is essential that we establish whether these benchmark test instances have the kind of properties that can support valid conclusions; namely that they are demonstrably diverse, unbiased, representative of intended applications, discriminating and challenging for a wide variety of algorithms. If we establish that the inherited benchmarks are lacking in any of these properties, we need methods to generate new test instances with the required characteristics to support rigorous performance evaluation (Hall and Posner, 2010), and drive new algorithm development. It is clear that new methodologies to evaluate both the suitability of test instance benchmarks, and the comparative performance of algorithms – more insightful and nuanced than simple "on average" reporting – are still needed, despite calls for the development of a more empirical science of algorithms over 25 years ago (Hooker, 1994).

In recent years, a new approach – known as Instance Space Analysis (Muñoz et al., 2018; Muñoz and Smith-Miles, 2020; Smith-Miles et al., 2014; Smith-Miles and Bowly, 2015) – has been proposed to answer this call. Test instances from a variety of sources, whether they are randomly generated, real-world, or classical benchmarks, can be visualised in a 2*D* projection of the entire space of possible test instances. The instance space is constructed by summarising each test instance as a high-dimensional feature vector of metrics that capture the intrinsic hardness of a test instance using a combination of problem dependent and independent difficulty measures (Smith-Miles and Lopes, 2012). Utilising the upper and lower bounds of each feature, the boundary of the possible test instance space can be projected to a 2*D* plane using dimension reduction methods, and the location of existing test instances can be scrutinised in the instance space to establish their diversity, unbiasedness and real-world-likeness. Super-imposing algorithm performance metrics across the instance space, offers

* Corresponding author.
E-mail addresses: smith-miles@unimelb.edu.au (K. Smith-Miles), munoz.m@unimelb.edu.au (M.A. Muñoz).

https://doi.org/10.1016/j.cor.2020.105184
0305-0548/

the opportunity to infer, using machine learning methods applied to the experimental data, the broader region where good performance can be statistically expected from each algorithm. This region is known as the *algorithm footprint*, and its area is an objective measure of comparative algorithmic power across the broadest possible test instance space. Unique regions of strength and weakness can be identified for each algorithm, and the features defining these regions can be explored to gain valuable insights into the conditions under which each algorithm is expected to perform well or poorly. Furthermore, this view of the available test instances offers the opportunity to recognise where the current benchmarks provide inadequate coverage of the instance space, and where the generation of new test instances would significantly augment our ability to understand algorithm performance across a wide range of test scenarios. Locating target points in the instance space where no test instances currently exist, evolutionary algorithms can be adopted to fill the instance space with evolved test instances with controllable properties, in a manner that is often not possible to achieve by manipulating instance generator parameters (Muñoz and Smith-Miles, 2020; Smith-Miles and Bowly, 2015).

Instance Space Analysis has now been successfully applied to a wide variety of combinatorial optimisation problems (Smith-Miles et al., 2014; Smith-Miles and Lopes, 2012), as well as continuous optimisation (Muñoz and Smith-Miles, 2017, 2020), supervised classification (Muñoz et al., 2018), time series forecasting (Kang et al., 2017), and anomaly detection (Kandanaarachchi et al., 2019). It is applicable to any field where there are algorithms developed and evaluated on suites of test instances, and the characteristics of those test instances can be adequately described with features that suggest their intrinsic hardness. The tools to support such analysis and insights are publicly available as a MATLAB toolbox (Muñoz et al., 2020), and with a web-based user interface known as MATILDA (Smith-Miles et al., 2019).

In this paper we apply Instance Space Analysis for the first time to provide new insights into the 0–1 Knapsack Problem (0–1KP). This binary optimisation problem involves deciding which of a finite set of items should be included in a knapsack, given the weight and profit of each item, with the goal to maximize the profit of the filled knapsack while meeting a fixed capacity constraint on its total weight. It has long been acknowledged that the classical instance classes used to test and compare algorithms for 0–1KP cover only a limited portion of the potential problem space, and many of these instance classes contain instances that are not considered difficult to solve for most algorithms (Pisinger and Toth, 1998; Pisinger, 2005; Hill and Reilly, 2000). As the algorithms designed to solve 0–1KP have grown in sophistication, and computational resources have become more powerful, the need for more difficult instance classes to augment the classical test sets and illustrate the strength of new algorithms has become more pressing (Martello et al., 1999). More difficult instances can be produced from classical instance classes by increasing the number of items and the magnitudes of the coefficients (Pisinger, 2005). However, this approach to generating harder test instances classes does not adequately test an algorithm's ability to deal with instances that are relatively small but are difficult to solve on account of their structural properties.

In response to this inadequacy of the classical instance classes, several new ones have been proposed in the last two decades which seek to explore the problem space more thoroughly. Martello et al. (1999) brought together and refined instance classes and ideas from earlier studies by Amado and Barcia (1993), Martello and Toth (1997), Pisinger and Toth (1998), Chvátal (1980) and Pferschy et al. (1997), to more comprehensively test the capabilities of the powerful COMBO algorithm (Martello et al., 1999). Hill and Reilly (2000) studied the complex interplay between the profit and weight correlations, and the resulting correlation structure between the objective function and the constraint slackness, demonstrating the impact on algorithm performance when this is varied beyond random correlations. Following this, in 2005 Pisinger (Pisinger, 2005) designed several new instance classes which frequently produce relatively difficult instances for all known algorithms.

A primary goal of the Instance Space Analysis presented in this paper is to improve our understanding of the underlying similarities between the instance classes proposed by Pisinger and the harder classical instance classes; in essence, *why* these instances are particularly hard. We also investigate the differences between hard instance classes which make them difficult in different ways, or to varying degrees for different algorithms. The diversity of all published 0–1KP test instance suites is explored via an instance space construction, and a new set of test instances are generated to fill the instance space. In this manner, we provide a visual answer to Pisinger's 2005 question: "Where are the hard knapsack problems?".

In order to construct an instance space for 0–1KP, this paper builds upon previous efforts to identify features summarising instance difficulty, much of which has already been exploited by previous research on automated algorithm selection methods for 0–1KP (Hall and Posner, 2007). In addition to drawing upon the established literature, we also propose and test some new features of 0–1KP that help to explain algorithm performance. Earlier efforts to characterise the difficulty of 0–1KP can be traced back to 1980 when Balas and Zemel (Balas and Zemel, 1980) defined a measure of knapsack difficulty based on a combination of the gap between the optimal and linear relaxation solutions and the range of item efficiencies. Chung et al. (1988) found that for a specific formulation of the strongly-correlated instance class the difficulty is strongly affected by several key features of instances: the weight of the least heavy item, the quantity added to each item's weight to define its profit, and the capacity of the knapsack. Hall and Posner (2007) also identified several significant features and applied them in an algorithm selection framework to decide between a branch-and-bound algorithm and a dynamic programming algorithm, the resulting meta-algorithm being termed KPCHOICE. Our selected features, based on the existing literature and some novel metrics, will be described later in this paper.

The remainder of this paper is structured as follows. In Section 2 we present the Instance Space Analysis framework as applied to the 0–1KP. Specifically, we discuss the 0–1KP problem in its general form, before describing the various classes of test instance benchmarks considered in this paper. We present three state-of-the-art algorithms to illustrate the comparative study, and the measures used to evaluate their performance. Finally, the experimental dataset – known as the "meta-data" – is completed with the calculation of a comprehensive set of features to describe the test instances. Once the rationale for these features is presented at the end of Section 2, the entire set of meta-data for the 0–1KP study is completely described. In Section 3 we construct an initial instance space using this meta-data, enabling the existing benchmark test instances to be projected and visualised in a $2D$ plane. We demonstrate where the hard test instances are located for various instance classes found in the literature, and assess the diversity of the suite of test instances within the theoretical boundary of the instance space. Considering algorithm performance, we then describe and predict the algorithm footprints for our three chosen algorithms. Combining these machine learning efforts, we are also able to perform automated algorithm selection to identify which algorithm is recommended for different regions of the instance space. The opportunity to generate additional test instances to fill the instance space is the focus of Section 4, where several methods are introduced to achieve a more comprehensive set of test

instances that reach to the boundaries of the instance space and fill interior gaps. The augmented meta-data, with the original benchmarks and our newly generated test instances, is then used to construct a more comprehensive instance space in Section 5 from which conclusions are drawn in Section 6.

2. Instance Space Analysis

The foundations for Instance Space Analysis (ISA) are firmly grounded in the Algorithm Selection Framework developed by Rice (1976) and inspired by the No-Free Lunch theorems of Wolpert and Macready (1997). Central to the methodology is the construction of an *instance space* whereby test instances are represented as points in a 2*D* plane, with the region of predicted good performance of an algorithm being its *footprint*. Moreover, through ISA we can identify regions where additional test instances may be required. New instances with controllable properties can be generated by targeting these regions, effectively "stress-testing" an algorithm under all possible conditions (Smith-Miles and Bowly, 2015).

The framework underpinning ISA is illustrated in Fig. 1. At its core there are six component *spaces*, the first of which is the ill-defined *problem space*, $\mathscr{P}$, containing all possible 0–1KPs. We assume the existence of a subset of instances, $\mathbf{I}$, for which we have computational results known as meta-data. Each instance $x \in \mathbf{I}$ is represented as a vector $\mathbf{f}_x$ in a *feature space*, $\mathscr{F}$, where each dimension corresponds to a measure of a mathematical or statistical property that correlates in some way to the performance of various algorithms. Next is the *algorithm space*, $\mathscr{A}$, comprising a portfolio of available algorithms to solve the instances in $\mathbf{I}$. The *performance space*, $\mathscr{Y}$, consists of a performance metric $y_{\alpha,x}$ of an algorithm $\alpha \in \mathscr{A}$ when solving a problem instance x, which could be based on the computational cost incurred while obtaining a satisfactory solution, or the quality of a solution obtained for a fixed computational budget. By projecting the feature data into 2*D* using dimension reduction methods (perhaps principal component analysis, or customised linear algebra and optimisation methods as proposed in Muñoz et al. (2018) to assist visualisation), we obtain the *instance space*, which allows the visual inspection of trends in algorithm performance and instance features. Through inspection of the instance space, we are able to gain insights into the kinds of instances that are well suited to each algorithm's strengths, and those that create challenges and expose weaknesses. Moreover, the mapping $g(\mathbf{f}_x, y_{\alpha,x})$ is learned from the meta-data to predict the performance of α on new instances, supporting automated algorithm selection.

It should be noted that the ISA methodology is *iterative*. That is, an initial instance space is created and explored based on currently available meta-data. From this fist iteration we may identify gaps in the instance space where new test instances need to be generated (Smith-Miles and Bowly, 2015). Once these new instances are added to $\mathscr{I}$, a different set of features may best describe the algorithms' performance and the 2*D* axes defining the instance space are likely to change. This entire process can be repeated until convergence, in other words, when the generated instances fully occupy the interior of the instance space boundary, and the features explaining variations in algorithm performance in a insightful manner have stabilised. In the following sections, we describe the details of the meta-data collected for construction of the initial instance space.

2.1. Problem space

The 0–1 Knapsack Problem (0–1KP) is defined as follows: Consider a set of n items with integer profits $\{p_1, \ldots, p_n\}$ and integer weights $\{w_1, \ldots, w_n\}$. We assume without loss of generality that

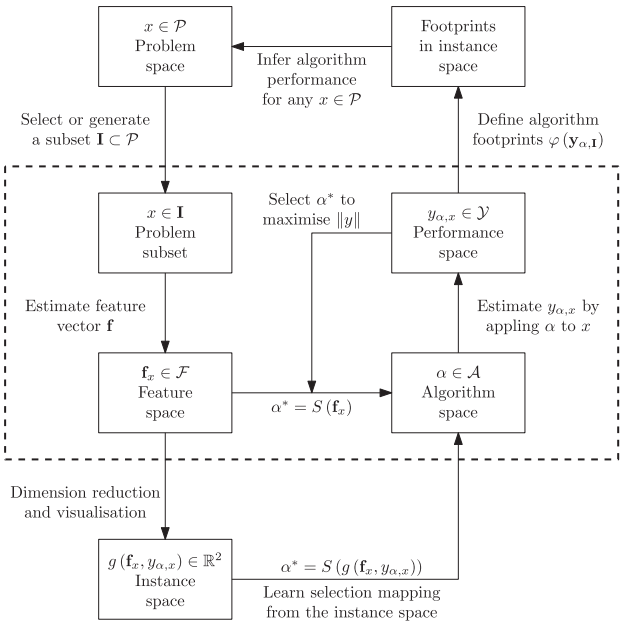

Fig. 1. Summary of the Instance Space methodology proposed by Smith-Miles et al. (2014), underpinned by the Algorithm Selection Problem framework (in the dotted box) of Rice (Rice, 1976).

all profits and weights are positive. The goal is to select a subset of items such that the sum of its profits is maximised while the total weight does not exceed the capacity of the knapsack, c. Expressed as an integer programming problem, the 0–1KP formulation is:

$$\text{maximize} \sum_{j=1}^{n} p_j x_j$$

$$\text{subject to} \sum_{j=1}^{n} w_j x_j \leqslant c$$

$$x_j \in \{0,1\},\ j = 1, \ldots, n$$

where x_j is a binary variable that determines whether item j is included in the knapsack, i.e., if $x_j = 1$ then item j is included, otherwise the item is not included. Other variations of the Knapsack Problem (KP) permit multiple copies of items, however the 0–1KP is a binary decision problem to determine if a single copy of an item is selected for inclusion in the constrained knapsack.

The efficiency of an item is defined as its profit divided by its weight. If the items in a given KP instance are sorted in order of decreasing efficiency, we can easily obtain upper and lower bounds on the value of the optimal solution by the following procedures. A lower bound on the optimal value of a given KP instance is easily obtained by finding any feasible solution. We can do this by applying the greedy algorithm: add items to the knapsack in order of decreasing efficiency, skipping any items which do not fit in the remaining capacity. An upper bound on the optimal value of a given KP instance can be obtained by finding the optimal value of a relaxation of the problem; specifically, the *linear relaxation* (in which each x_j is in $[0,1]$ rather than $\{0,1\}$. We can find the optimal solution of this relaxation by adding items to the knapsack in order of decreasing efficiency, then when we find an item which does not fit in the remaining capacity, add a fraction of that item such that the knapsack is completely filled. This last item is called the *split item*. Since all feasible solutions to an integer KP instance have integer value, an improved upper bound may be obtained by

rounding down the upper bound obtained from the linear relaxation, if it is not already an integer.

An instance of 0–1KP may be altered by requiring that a particular item i must or must not be included in the knapsack packing. The resulting problem may also be expressed as a knapsack problem by removing item i from the item set and decreasing the knapsack capacity by w_i if the item must be included. If the item must not be included there is no change in the knapsack capacity. This property of KP is very useful as it allows us to obtain insights into the optimal solution for a particular instance by solving smaller, related instances. For instance, if the lower bound of the original knapsack problem is greater than or equal to the upper bound obtained after "fixing" an item as in or out of the packing, there are no solutions with this fixing which improve on the known lower bound solution, and so we can safely fix this item in the *opposite* way, reducing the number of variables which need to be actively considered.

An important concept for the algorithms which we compare in this paper is the *core problem*, which considers only a core subset of items normally having similar efficiency to the split item. If the number of items included in the core problem is adequate, an optimal solution to the original instance can be obtained by fixing all variables outside the core subset based solely on their respective items' efficiencies, then solving the core problem. The algorithms under consideration apply various methods to finding and solving this core problem. Unfortunately, it is difficult to predict the minimal size for the core problem which will be adequate to ensure (certainly or with high probability) that an optimal solution for the overall problem will be found (Pisinger, 1995). Furthermore, when the distribution of item weights in the core problem is not uniform, the core problem may need to be larger to obtain good solutions and need not be particularly easy to solve (Pisinger, 1999a). Since the latter phenomenon is driven by instance structure, when assessing the performance of algorithms based on solving the core problem it is important to understand the character of the instances being used to test those algorithms.

2.2. Instance subset

KP instance classes are typically defined with respect to a data range parameter (denoted herein by R) which limits the magnitude of the weight and profit coefficients, to ensure a fair comparison between different instance classes. In this paper we will consider instances with $n = 1000$ items and data range $R = 1000$ (note however that the construction method for some instance classes results in individual item profits greater than R). We generate 200 instances from each instance classes summarised in Table 1. Many of these instance classes are derived from those used in Martello et al. (1999) and Pisinger (2005), in which detailed descriptions can be found. Our specific generation procedures for the instance set[1] are as follows.

Of the 200 instances generated for each instance class, the capacity of instance number k is set to $\lceil \frac{k}{201} \sum_{j=1}^{n} w_j \rceil$ (based on the item weights generated for that instance). This is intended to ensure that a wide range of reasonable capacities is explored by the instance space. In several classes which would normally use constant parameters (frequently $R/10$) to generate each item's profit from its weight or vice versa, we instead generate a random offset from an interval including the typical value. For example, the ordinary procedure for generating an Almost Strongly Correlated class randomly generates the unit weights with a discrete uniform distribution in $[1, R]$, then generates each item i's profit in the range $[w_i + \Delta - \epsilon, w_i + \Delta - \epsilon]$ using the constants $\Delta = R/10$ and $\epsilon = R/500$. We instead randomly choose Δ and ϵ independently for each instance in the ranges $[R/20, 3R/20]$ and $[R/500, 3R/500]$ respectively. Similar alterations have been made to all of the instance classes whose definitions in Table 1 include a parameter Δ. In general, choosing different parameters of similar magnitude in this way does not substantially alter the character of these instance classes.

A more substantial alteration has been made to the Similar Uncorrelated Weights instance class. As in the normal generation procedure the item profits are randomly generated with a uniform random distribution in $[1, R]$; however, instead of randomly generating item weights in the range $[100000, 100100]$, we generate the item weights in the range $[R - \Delta, R]$ instead, with Δ randomly chosen for each instance in the range $[R/20, 3R/20]$ as before. This is intended to facilitate a fairer comparison between instance classes, although it does significantly change the character of these instances; in particular it no longer satisfies the definition of "very similar coefficients" specified by Amado and Barcia (1993) in the original proposition of this class. The Spanner-type classes are generated as in Pisinger (2005) by generating a pair of "spanner set" items, normalising them, and adding items by taking integer multiples of the spanner set items. However for each instance we randomly generate a multiplier limit m between 5 and 15 rather than using a fixed multiplier limit. Furthermore, we divide the spanner item coefficients by m rather than $m/2$ to ensure that the final item weights do not exceed R.

We also include in our instance subset several classes not found in previous papers. These classes were motivated by filling holes and pushing boundaries in early instance space experiments not presented in this paper. The *Quadratic Fit* and *Cubic Fit* instances are generated by the following procedure. First, we create $n/10$ temporary "dummy" items by choosing their weights and profits with a uniform random distribution in $[1, R]$ (as per the Uncorrelated instances). Second, we find the quadratic or cubic curve (with item weights as the independent variable) which best fits these points in a least-squares sense. Finally, we create the actual items for the final knapsack problem by choosing their weights and profits with a uniform random distribution in $[1, R]$, then substituting into the quadratic/cubic equation obtained in the previous step to obtain the profit for each item (with a minimum profit of 1 and rounding up to the nearest integer). The *Random Ceiling* instances are generated by choosing a profit increment q_j randomly from $\{0, 1, 2\}$ for all $j \in \{1, \ldots, R\}$. We set each item's weight w_i to a random integer between 1 and R and each item's profit $p_i = w_i + q_{w_i}$. This ensures that items with the same weight also have the same profit value, similar to the Profit Ceiling class. The *Profit Ceiling Large Only* instance class generation procedure sets each item's weight w_i to a random integer between $R/2 + 1$ and R, and assigns profits in the same way as the Profit Ceiling class ($p_i = 3\lceil w_i/3 \rceil$).

2.3. Algorithm space

The portfolio of algorithms considered in the meta-data comprises EXPKNAP, MINKNAP and COMBO to define the algorithm space, $\mathscr{A}$, of which the latter two are considered state-of-the-art (Pisinger and Saidi, 2017).

EXPKNAP (Pisinger, 1995) is a branch-and-bound algorithm which builds a growing core about the split item. Items are tested to see if they are a good choice for the core or rejected, and the core size expands, testing new items until an optimal solution is found.

MINKNAP (Pisinger, 1997) is a dynamic programming-based algorithm which also builds a core around the split item, but places more emphasis on ensuring that the size of the core is minimal.

[1] Available for download from https://matilda.unimelb.edu.au/matilda/problems/opt/knapsack.

Table 1
Instance classes used in the meta-data, showing their source (* denotes new instance classes) and generation parameters. $[a, b]$ signifies a random integer between a and b with uniform distribution. For each applicable instance the following quantities are chosen at random: $\Delta \in [\frac{R}{20}, \frac{3R}{20}]$, $\epsilon \in [\frac{R}{500}, \frac{3R}{500}]$, $f \in [3, 10]$.

Instance family	w_j	p_j	Balas and Zemel (1980)	Martello and Toth (1988)	Martello and Toth (1997)	Pisinger (1999b)	Pisinger (1995)	Pisinger (1997)	Martello et al. (1999)	Pisinger (2005)	[*]
Uncorrelated	$[1, R]$	$[1, R]$	✓	✓	✓	✓	✓	✓	✓	✓	
Weakly Correlated	$[1, R]$	$[w_j - \Delta, w_j + \Delta]$	✓	✓	✓	✓	✓	✓	✓	✓	
Strongly Correlated	$[1, R]$	$w_j + \Delta$	✓	✓	✓	✓	✓	✓	✓	✓	
Subset Sum	$[1, R]$	w_j	✓	✓	✓	✓	✓	✓	✓	✓	
Inverse Strongly Correlated	$p_j + \Delta$	$[1, R]$			✓	✓			✓	✓	
Almost Strongly Correlated	$[1, R]$	$[w_j + \Delta - \epsilon, w_j + \Delta + \epsilon]$			✓	✓			✓	✓	
Similar Uncorrelated Weights	$[R - \Delta, R]$	$[1, R]$				✓			✓	✓	
Even–Odd Strongly Correlated	$[1, R/2] \times 2$	$w_j + R/10$							✓		
Even–Odd Subset Sum	$[1, R/2] \times 2$	w_j							✓		
No Small Weights	$[R/2 + 1, R]$	$[w_j - \Delta, w_j + \Delta]$							✓		
Spanner Uncorrelated	span. size $= 2$	mult. limit $\in [5, 15]$, see text								✓	
Spanner Weakly Correlated	span. size $= 2$,	mult. limit $\in [5, 15]$, see text								✓	
Spanner Strongly Correlated	span. size $= 2$,	mult. limit $\in [5, 15]$, see text								✓	
Multiple Correlated	$[1, R]$	$w_j + 3R/10$ if $\mathrm{mod}(w_j, f) = 0$, $w_j + 2R/10$ otherwise								✓	
Profit Ceiling	$[1, R]$	$3\lceil w_j/3 \rceil$								✓	
Circle	$[1, R]$	$2/3\sqrt{4R^2 - (w - 2R)^2}$								✓	
Quadratic Fit	$[1, R]$	$aw_j^2 + bw_j + c$, see text									✓
Cubic Fit	$[1, R]$	$aw_j^3 + bw_j^2 + cw_j + d$, see text									✓
Random Ceiling	$[1, R]$	See text									✓
Profit Ceiling Large Only	$[R/2 + 1, R]$	$3\lceil w_j/3 \rceil$									✓

The algorithm expands out through the items in the instance in a cyclic pattern of growth and culling, periodically rechecking the core against boundaries and rejecting unpromising items.

COMBO (Martello et al., 1999) is also a dynamic programming-based algorithm which combines elements from MINKNAP and Martello and Toth (1997), as well as several new ideas. As compared with MINKNAP, the COMBO algorithm generates the starting core problem in a more sophisticated manner, employs surrogate relaxation of cardinality constraints to obtain upper bounds, may briefly examine items outside the core to improve known feasible solutions, and checks for situations where the item weights share a common multiplicative factor which may be removed.

2.4. Performance space

Since all of the algorithms in the preceding section solve 0–1KP to optimality, our performance metric is the wall-clock time required for each algorithm to obtain and verify the optimal solution for the instance when run on the Spartan HPC system (Meade et al., 2017), using one virtual machine 2.1 GHz core with 8 GB RAM for each test. Appendix A presents statistical performance of the CPU times for the three algorithms. If for a given algorithm and instance this time is less than 0.1 s, we run the algorithm multiple times and take the average runtime to obtain a reliable estimate. In cases where EXPKNAP fails to solve a given instance in less than 15 s, its performance metric for that instance has been arbitrarily set to 15 s. Since this is nearly two orders of magnitude more time than COMBO requires to solve any instance in our instance subset, this serves as a reasonable placeholder for failure.

For any given instance the ISA toolkit defines an algorithm as having "good" (or "bad") performance if its performance metric falls with (or outside) a user-defined tolerance percentage compared to the best-performing algorithm. For the purposes of this paper we defined an algorithm's performance as "good" if it requires at most 20% more time to obtain the optimal solution compared to the best performing algorithm in the portfolio, and "bad" if this condition is not satisfied.

2.5. Feature space

The features which compose our feature space are presented in Table 2. Although the instances we study in this paper all have a

Table 2
Description of 13 out of 23 feature concepts used in our 0–1KP meta-data

Feature Name	Bound	Description
1. Dominant Pairs	$[0, 1]$	Proportion of item pairs i,j for which $p_i \geqslant p_j$ and $w_i \leqslant w_j$ OR $p_j \geqslant p_i$ and $w_j \leqslant w_i$, excluding identical items. Similar to Hall and Posner (2007).
2. Smaller Better Pairs	$[0, 1]$	Proportion of item pairs i,j for which $p_i/w_i > p_j/w_j$ and $w_i \leqslant w_j$ OR $p_j/w_j > p_i/w_i$ and $w_j \leqslant w_i$.
3. Capacity Fraction	$[0, 1]$	Capacity of the knapsack divided by the sum of the weights of all items.
4. Capacity Fraction Distance From 1/2	$[0, 1]$	$\lvert$Capacity Fraction $- 0.5\rvert \times 2$
5.Correlation Coefficient	$[-1, 1]$	Correlation coefficient of item weights and profits, viewed as points (w_i, p_i) in $\mathbb{R}^2$
6. Approximation Gap	$[0, 1]$	Difference between the linear relaxation upper bound and greedy algorithm lower bound (similar to Hall and Posner (2007)) divided by the profit assigned to the most profitable item.
7.Coefficient of Variation of Weights	$[0, 1]$	Standard deviation of item weights divided by average item weight, as in Hall and Posner (2007). Normalised as per (1) with $D = 1$.
8. Coefficient of Variation of Profits	$[0, 1]$	Standard deviation of item profits divided by average item profit, as in Hall and Posner (2007). Normalised as per (1) with $D = 1$.
9. Coefficient of Variation of Efficiencies	$[0, 1]$	Standard deviation of item efficiencies divided by average item efficiency. Normalised as per (1) with $D = 5$.
10. Possible Item Fix Proportion	$[0, 1]$	Proportion of items which can be determined to be included or excluded in the optimal packing of the knapsack by fixing them as in or out of the knapsack and comparing the value of the linear relaxation solution of the resulting problem with the value of the greedy solution of the original problem.
11. Subset Sum Likeness	$[0, 1]$	Proportion of items with efficiency within 0.05% as the split item. Intended to measure the similarity of the instance to a Subset Sum instance.
12. Even–Odd Likeness	$[0, 10]$	Equal to Subset Sum Likeness, but multiply by 10 if the weights of the items that have the split item's efficiency have a greatest common divisor greater than 1. Intended to measure the similarity of the instance to an Even–Odd Subset Sum instance, or a Profit Ceiling instance with sufficiently large capacity.
13. Modified Balas-Zemel Measure	$[0, 1]$	Modified version of the measure defined in Balas and Zemel (1980) using the value of the greedy solution in the place of the value of the integer optimum solution: $\frac{\text{Linear Relaxation value} - \text{Greedy Solution value}}{(1/2)max_{i\in\{1,\ldots,N\}}\lvert p_i - (w_i \times \text{Efficiency of Split Item})\rvert}$Normalised as per (1) with $D = 500$.
14. Maximum Cardinality	$[0, 1]$	Upper bound, defined in Martello and Toth (1997) Eq. (11), on the number of items which may potentially be included in the optimal solution, divided by n.
15. Minimum Cardinality	$[0, 1]$	Lower bound, defined in Martello and Toth (1997), on the number of items which may potentially be included in the optimal solution found by trying to add items in descending order of weight, divided by n.
16. Cardinality Gap	$[0, 1]$	Difference between Maximum Cardinality and Minimum Cardinality.
17. First Weight	$[0, 1]$	Weight of the least heavy item divided by the weight of the heaviest item. Similar feature used in Chung et al. (1988).
18. First Profit	$[0, 1]$	Profit of the least profitable item divided by the weight of the most profitable item.
19. Polyfit Linear	$[0, 1]$	Consider the items as defining points (w_i, p_i) in $\mathbb{R}^2$, and find the linear equation $P(w) = aw + b$ which best fits these points in a least-squares sense. Define $\bar{p} = \frac{1}{n}\sum_{i=1}^{n} p_i$. Then the value of the feature is $\max\left\{1 - \frac{\sum_{i=1}^{n}(p_i - P(w_i))^2}{\sum_{i=1}^{n}(p_i - \bar{p})^2}, 0\right\}$
20. Polyfit Quadratic	$[0, 1]$	As Polyfit Linear, but considering a quadratic equation $P(w) = aw^2 + bw + c$.
21. Polyfit Cubic	$[0, 1]$	As Polyfit Linear, but considering a cubic equation $P(w) = aw^3 + bw^2 + cw + d$.
22. Before/After Split Ratio	$[0, 1]$	Let the number of items with efficiency strictly better than that of the split item be a, and the number of items with efficiency equal to or worse than that of the split item be b. If $a > b$ then the feature is defined as $\frac{\arctan(\frac{a}{b}-1)}{\pi} + \frac{1}{2}$. Otherwise it is defined as $\frac{\arctan(1-\frac{b}{a})}{\pi} + \frac{1}{2}$
23. Greedy Unused Capacity	$[0, 1]$	If the instance has no item pairs i,j such that $w_i \neq w_j$ then this feature defaults to 0. Otherwise it is defined as the capacity unused by the greedy solution, divided by the smallest non-zero difference between any two items' weights, then normalised as per (1) with $D = 100$.

similar number of items and range of coefficients, where appropriate we have scaled the features so that they can be applied to instances where these assumptions do not apply. In cases where the natural interpretation of a feature is unbounded, or the typical range of variation in the feature with respect to our instance subset is considerably smaller than the theoretical range, we apply the following normalisation process:

$$\text{normalised feature} \leftarrow \frac{\tan^{-1}(\text{raw feature value}/D)}{\pi/2} \tag{1}$$

The value of D is specified independently for each feature where this normalisation is applied. In each case we have chosen D so that the mapping in (1) is reasonably close to linear across the range of feature values which are typical with respect to our instance subset. Almost all of these features are also applied to the reduced knapsack. The Possible Item Fix Proportion for the reduced knapsack is almost always zero and the Even–Odd Likeness may be misleadingly high if the reduced knapsack is small, so these features are excluded. This means that in total 44 features are measured for each knapsack instance based on the 23 feature concepts described in Table 2.

The choice of features must be computationally tractable, but since our focus is insights rather than automated real-time algorithm selection, we permit features with greater computational complexity than real-time automated algorithm selection can afford. In particular, we require sorted lists of the items, based on efficiency, profit and weight, in order to calculate many of the features, which the 0–1KP algorithms under consideration go to some effort to avoid. Once this sorting is performed once for a given instance, the only features with time complexity greater than $O(n)$ are the Dominant Pairs and Smaller Better Pairs features. The worst-case time complexity of calculating these features is $O(n^2)$, but by dividing the (weight, profit) space into a grid of smaller rectangles and identifying how many items are in each rectangle, we can avoid many trivial item-pair comparisons and for reasonable 0–1KP instances obtain far better practical performance than the naive approach of comparing each pair. We do avoid the use of features which rely upon knowledge of the optimal solution of an instance.

3. Constructing an initial instance space

The Instance Space Analysis Toolkit is a MATLAB based set of tools that facilitate the construction of an instance space (Muñoz et al., 2020). The toolkit contains both an automated data processing pipeline, and functions that perform specific stages of the anal-

ysis. The implementation details for constructing an initial 0–1KP instance space are described in this section with full reproducibility enabled with the meta-data and code available from https://matilda.unimelb.edu.au/matilda/problems/opt/knapsack.

As an initial feature preprocessing step in the pipeline, each feature is bounded using the procedure described in Section 2.5, and then the variance is stabilised and distribution normalised using a one parameter Box-Cox transformation. Finally, each feature is standardised to $\mathcal{N}(0, 1)$ using a z-transformation.

The toolkit also provides routines for automatically selecting the most informative features, based on their correlation with algorithm performance. The top N most correlated features are considered for each algorithm, with some common to several algorithms. We select $N = 8$, creating a subset of no more than 8 features for each of our 3 algorithms. These features are then clustered, with a Pearson correlation used as a dissimilarity metric, to identify clusters of similar features. Taking one feature from each cluster, various candidate subsets of features are tested for how well they can discriminate between easy and hard instances. This is evaluated by projecting the instances to $2D$ using principal component analysis, and then using Random Forests, a machine learning classifier, to separate the instances labelled with "good" or "not good" performance based on the performance metric described in Section 2.4. The candidate subset of features that enables the Random Forest to achieve the best accuracy in predicting algorithm performance is chosen as the "optimal" subset of features. Using this feature selection process for our 0–1KP meta-data, eight features are selected by MATILDA: those numbered 1, 10, 12, 18 and 21 for the original 0–1KP instance, and those numbered 7, 9 and 23 for the reduced instance.

Finally, we use the Projecting Instances with Linearly Observable Trends (PILOT) method[2] Muñoz et al. (2018), to find a projection from $8D$ to $2D$, such that algorithm performance and feature values increase linearly from one edge of the instance space to the opposite, thereby assisting the visualisation of directions of hardness and feature correlation to support insights. To find the projection, we make use of BFGS (Broyden, 1970) as an optimisation algorithm. However, the underlying optimisation problem that PILOT solves is convex but highly ill-conditioned, resulting on infinite solutions spanning a line on the space. As such, the solution is dependent on the starting point for a non-stochastic algorithm such as BFGS. Therefore, we calculate 30 different projections and select the one with the highest topological preservation, defined as the correlation between high- and low-dimensional distances. Details of the method are provided in Muñoz et al. (2018).

Each instance x can now be projected from a point in an $8D$ feature space $f(x)$, to a point in a $2D$ instance space (Z_1, Z_2), via the optimised linear transformation found by the PILOT method, as summarized by Eq. (2).

$$\begin{bmatrix} Z_1 \\ Z_2 \end{bmatrix} = \begin{bmatrix} 0.1894 & 0.2337 \\ 0.0849 & 0.2904 \\ 0.3421 & -0.0741 \\ 0.0150 & -0.2787 \\ -0.2867 & -0.3419 \\ -0.2937 & 0.0156 \\ 0.2840 & -0.3769 \\ 0.2964 & -0.2181 \end{bmatrix}^{\mathrm{T}} \begin{bmatrix} \text{Dominant Pairs} \\ \text{Possible Fix Prop.} \\ \text{First Profit} \\ \text{Polyfit Cubic} \\ \text{Even} - \text{Odd Likeness} \\ \text{Red. Coeff. Var. Weights} \\ \text{Red. Coeff. Var. Effic} \\ \text{Red. Greedy Unused Capac.} \end{bmatrix} \tag{2}$$

[2] In previous work we referred to this method as Prediction Based Linear Dimensionality Reduction (PBLDR).

The two axes defining the instance space in the (Z_1, Z_2) plane are linear combinations of the 8 selected features. Projecting all instances in our meta-data set, Fig. 2 shows each instance as a point with its (Z_1, Z_2) coordinate location determined by Eq. (2). Different colors identify the various classes: instances proposed by Pisinger (2005) in brown; instances proposed in earlier papers in black; as well as the additional instance classes we have designed to add diversity to our initial meta-data (designated as 'Designed' instance classes to differentiate them from the evolved instances we will consider later) in orange. Further breakdown of the various sub-classes within these instance sets is shown in Fig. 3.

3.1. Where are the hard knapsack instances?

In order to answer the key question of where the hard knapsack instances lie in the instance space, we must inspect algorithm performance to identify the location of easy and hard regions, and determine which instances are providing challenge for algorithms. Fig. 4 shows the performance metric (run-time) for each of the three algorithms, with a color scale ranging from scaled minimum (black) to scaled maximum (orange) values. The scaling is based on a single algorithm's range of run-times for the left-side plots (i.e. relative to its own range), and the range of run-times across all algorithms for the right-side plots (i.e. relative to the portfolio range).

Superimposing the distribution of the 8 features across the instance space in Fig. 5 permits visual comparison of the properties of various instances based on their location. Combined with an inspection of algorithm performance in Fig. 4, we have all the necessary perspectives to draw insights from the Instance Space Analysis and answer the question of "where are the hard knapsack instances", and explain what makes them hard.

The EXPKNAP algorithm's performance on any given instance generally falls into one of two categories; either the instance is solved very quickly or not at all within the 15 s time limit. Most of the instances which are solved quickly lie in the upper part of the space (in the positive Z_2 direction). There are also a few clusters of instances in the lower-left part of the space which EXPKNAP solves successfully; most of these instances are in the Subset Sum or Random Ceiling instance classes. While EXPKNAP does solve some of these instances faster than either MINKNAP or COMBO, its superiority in these areas is far less important than its inability to solve many other instances in a reasonable timespan. As such its overall performance is not competitive with COMBO and MINKNAP.

For COMBO and MINKNAP the most difficult instances are found in the lower part of the instance space i.e. for negative values of Z_2, and on the left side or middle of the space. These 'difficult' areas are characterised by lower values of the Possible Item Fix Proportion and Dominant Pairs features. We would intuitively expect that instances in which few items are obviously 'better' than other items, and few items can be trivially fixed as in or out of the optimal packing, would be harder to solve. The Polyfit Cubic feature has large values across the 'difficult' area but also in areas of the instance space which are less difficult. Furthermore, within most of the instance classes the more difficult instances for COMBO and MINKNAP tend to be lower in the instance space, no matter whether the instance class as a whole is easy or hard.

A closer inspection of the instance space reveals that the 'difficult' area of the instance space has two distinct clusters with gaps in between that are occupied by only a few instances. In the following sections we will discuss the underlying properties of these clusters with a view to understanding the properties which distinguish instances in each cluster.

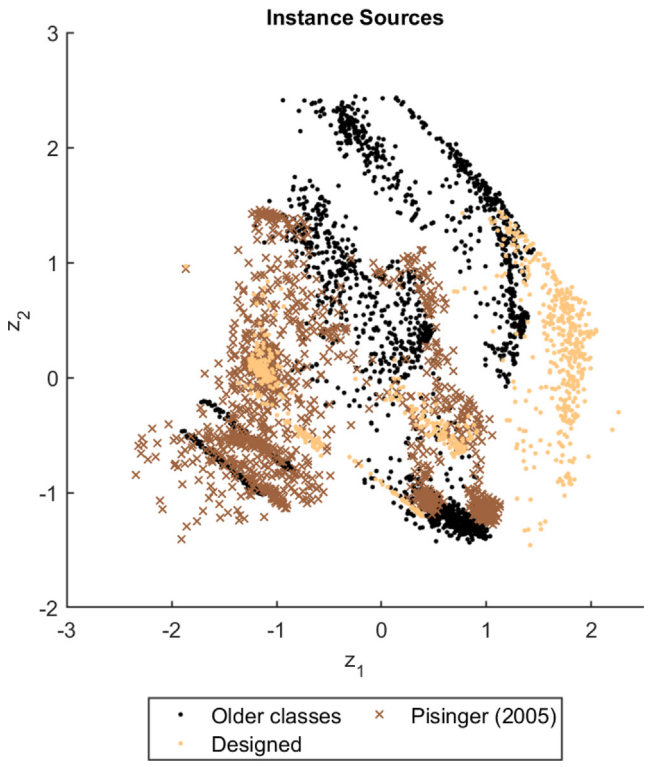

Fig. 2. Instance classes in the initial 0–1KP instance space, with axes given by Eq. (2).

3.1.1. Instances with strong correlation

The cluster of hard instances around $Z_1 = 1$ and $Z_2 = -1$ is characterised by a large coefficient of variation of efficiencies in the reduced knapsack. This implies that while these instances contain some items with significantly varying efficiencies, it is not trivial to determine whether or not many of these items belong in the optimal knapsack packing. These instances also typically have a high value of the Greedy Unused Capacity feature in the reduced knapsack, although they share this distinction with some Quadratic and Cubic Fit instances on the right-hand side of the space which are not as difficult.

The instances on the lower side of this cluster have the interesting property that while they are particularly difficult for MINKNAP, the COMBO algorithm finds many of them comparatively easy; this is most clearly illustrated in Fig. 6. These instances are primarily drawn from the Strongly Correlated, Almost Strongly Correlated and Even–Odd Strong instance classes. Since the COMBO algorithm was motivated by a desire to improve on the performance of earlier algorithms (including MINKNAP) when applied to Strongly Correlated instances, the relatively good performance of COMBO on these instances is not unexpected.

Some instances from the Circle and Multiple Strongly Correlated instance classes proposed by Pisinger (2005) form dense groups on the upper side of the strong-correlation cluster. Compared to the instances on the lower side they are similarly difficult for MINKNAP but harder for COMBO. This suggests that the underlying reasons why all of these classes are relatively difficult are similar, but that the less direct approach used by the Circle and Multiple Strongly Correlated instance classes to achieve this difficulty are not handled as well by the tools which distinguish COMBO from MINKNAP.

This cluster also contains a few instances from the Profit Ceiling Large Only instance class in a line which extends into the gap separating the two high-difficulty clusters. We defer further analysis of this group of instances until after an examination of the second cluster.

3.1.2. Instances with many similar items

The cluster of instances around $Z_1 = -1.5$ and $Z_2 = -0.5$ is primarily composed of instances from the Random Ceiling, Profit Ceiling, Subset Sum-type and Spanner-type instances. These instance classes share the property that each instance contains a large number of items with identical efficiencies, and are particularly characterised by high values of the Even Odd Likeness feature. However, there are significant differences between the instance classes in terms of their difficulty.

The Random Ceiling and Subset Sum instances are typically easy for all three algorithms. The Even–Odd Subset Sum instances are typically not solved within the 15 s time limit by EXPKNAP and are quite difficult for MINKNAP to solve, but are relatively easy for COMBO to solve. Many of the Spanner-type instances are fairly difficult for all three algorithms to solve; those which are easier (often from the Spanner Uncorrelated class) are found higher in the instance space, in the direction of the easier instances from other classes.

The Profit Ceiling instances are separated into two main groups; a larger cluster above a smaller line. The upper cluster contains instances which have lower capacity fractions, and are relatively easy for MINKNAP and COMBO. The lower line contains instances which have high capacity fractions, some of which are very difficult for both MINKNAP and COMBO. Notably the COMBO algorithm has a relatively small advantage over MINKNAP with respect to these instances.

The separation of these instance classes from the strong-correlation cluster reflects a clear difference in character between the two types of problems. The difficulty of the Even–Odd Subset Sum instances for MINKNAP and some of the Spanner-type and Profit Ceiling-type instances for both MINKNAP and COMBO are based not only on the large number of similar-efficiency items, but also upon special properties of the item weight and profit distributions which make optimality difficult to prove.

The Profit Ceiling Large Only instance class has a similar separation between a higher, easier cluster of low capacity fraction instances and a lower, often harder line of high capacity fraction instances. The hardest instances from this class are harder for COMBO than those in the Profit Ceiling class, or indeed any other instance class among those studied here. However, in the instance space the instances from this class are far to the right of the Profit Ceiling instances, and are in fact closer to the high-correlation cluster. This phenomenon is primarily caused by the Profit Ceiling Large Only instances having a large value of the First Profit feature, a distinction they share with the Quadratic and Cubic Fit instances on the far right-hand side of the space.

Since the Profit Ceiling Large Only instances are difficult not only by virtue of their structure (characteristic of the left-hand cluster) but also through simply having larger problem data coefficients (a more brute-force approach to difficulty) it does not seem entirely unreasonable that the Profit Ceiling Large Only instances should be somewhat separate in the instance space from the Profit Ceiling, Spanner and Even–Odd Subset Sum instances. However, we would not expect them to be so close to the high-correlation cluster, given that their problem structure and algorithm performance characteristics are quite different.

It is worthwhile to note that the increased difficulty of the Profit Ceiling Large Only instance class over the Profit Ceiling class may not be entirely 'fair'. As a side effect of altering the general distribution of the item weights and profits to remove small items, the Profit Large Only class increases the *average* item weight and hence the capacity of the knapsack (since this is defined as a fraction of the sum of item weights). Since we restricted our instances to have

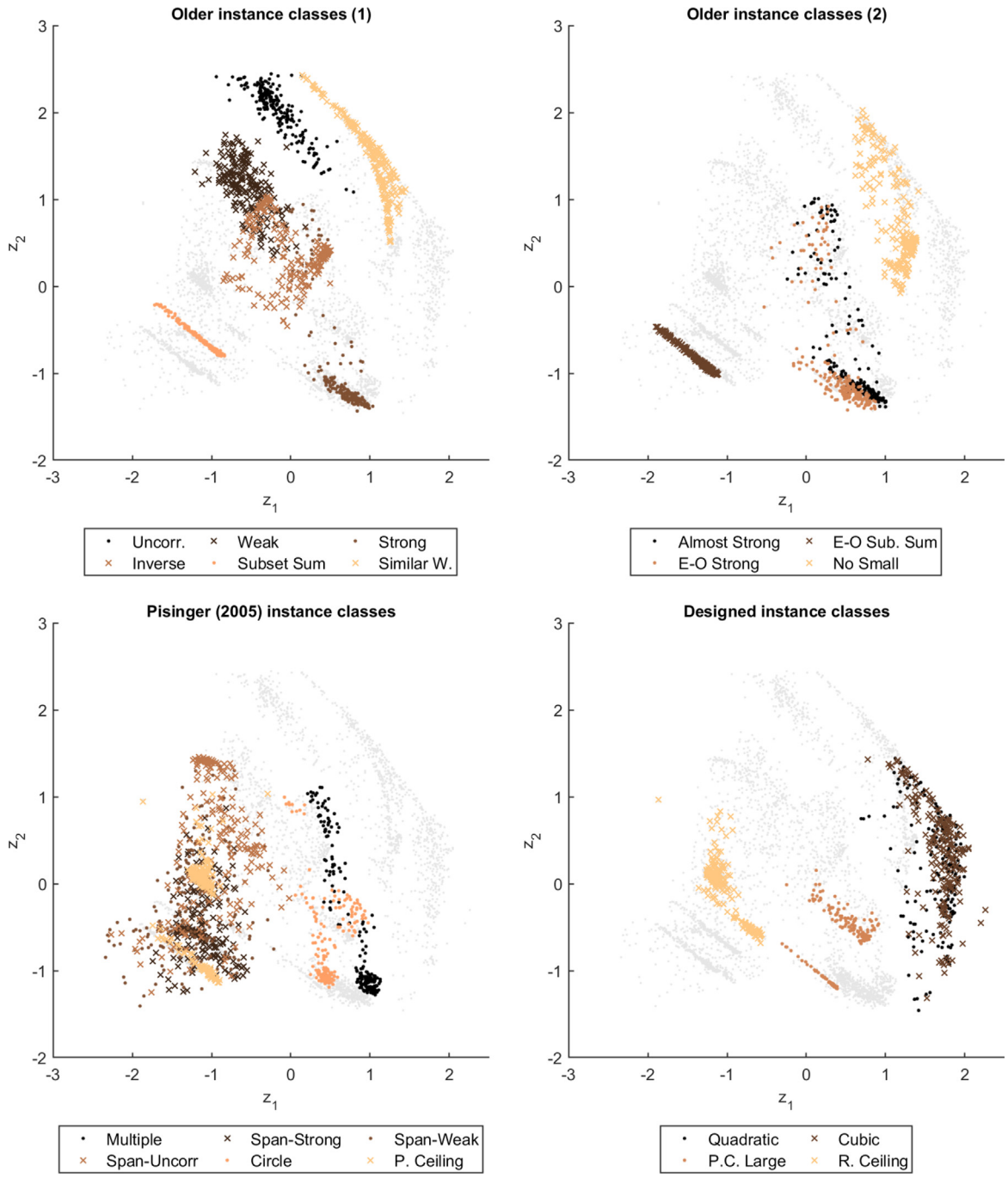

Fig. 3. Breakdown of various instance classes described in Table 1 within the initial 0–1KP instance space.

a maximum weight *per item*, as was done in Martello et al. (1999) and elsewhere, we observe that the Profit Ceiling Large Only instances are more difficult. If we instead restricted the maximum *knapsack capacity*, the Profit Ceiling Large Only type of structure might not result in harder instances than the Profit Ceiling class. Future investigation into the relative difficulty of knapsack instance classes should bear this potential source of bias in mind.

3.2. Learning algorithm footprints and automated algorithm selection

An advantage of visualising the instances within a 2*D* instance space is that we can observe "regions" of strength for each algorithm. Also, the space can be used for automated algorithm selection for untested instances. That is, given the coordinates of an untested instance, we can identify the algorithm most likely to perform best. For this task, we employ a machine learning model known as a support vector machine (SVM) with a polynomial kernel. For each algorithm, a SVM is trained with the instance space coordinates as the input, and the binary "goodness" performance metric described in Section 2.4 as output. The SVM is finely tuned using Bayesian Optimisation and validated using 5-fold cross-validation.

The results of the three trained SVMs are shown in Fig. 7, along with an automated algorithm selection recommendation. Each algorithm's SVM makes a binary prediction of whether the algorithm performance metric will be good or bad for each instance.

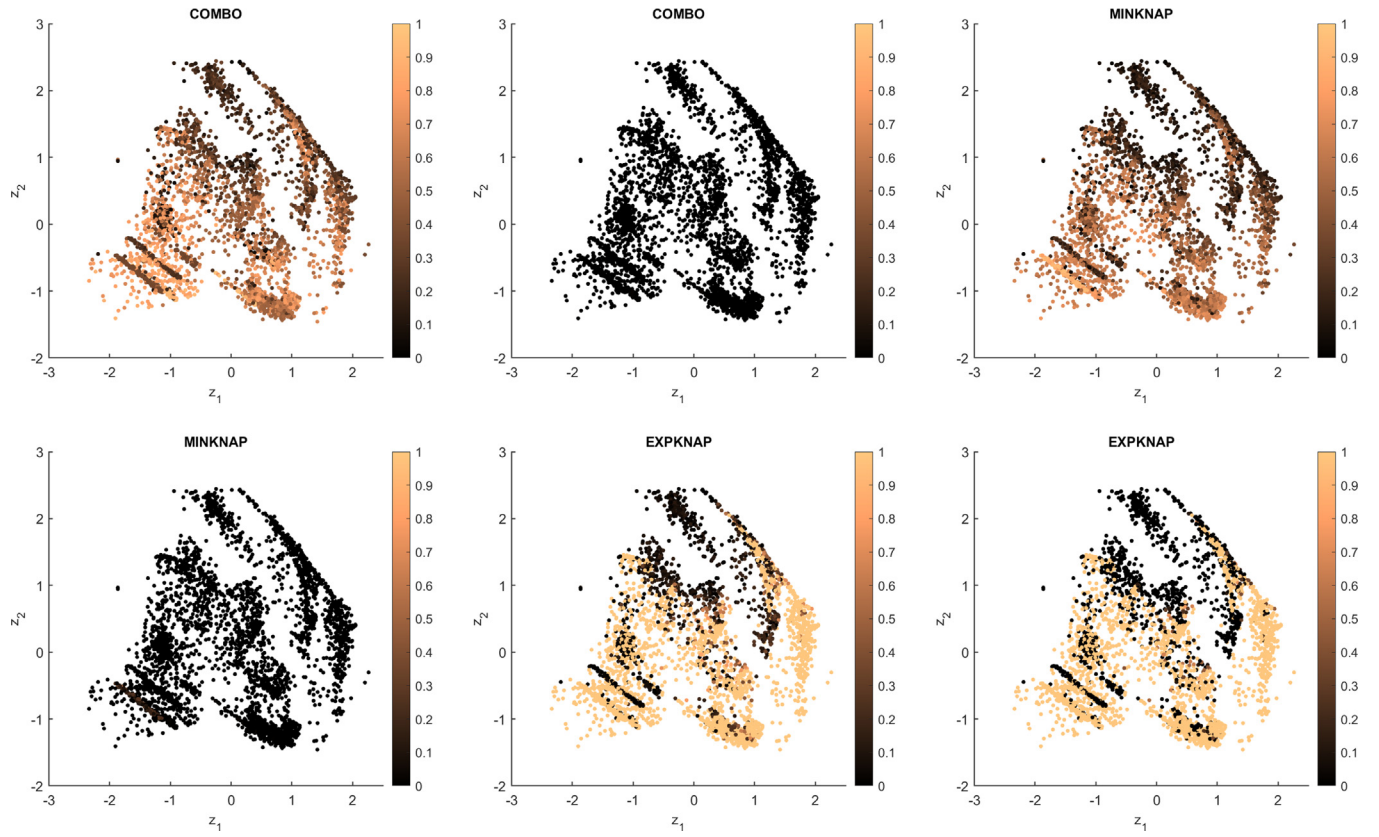

Fig. 4. Scaled algorithm performance metric (run-time) for each algorithm (per row): relative to its own performance range (left column), and relative to the algorithm portfolio's range (right column).

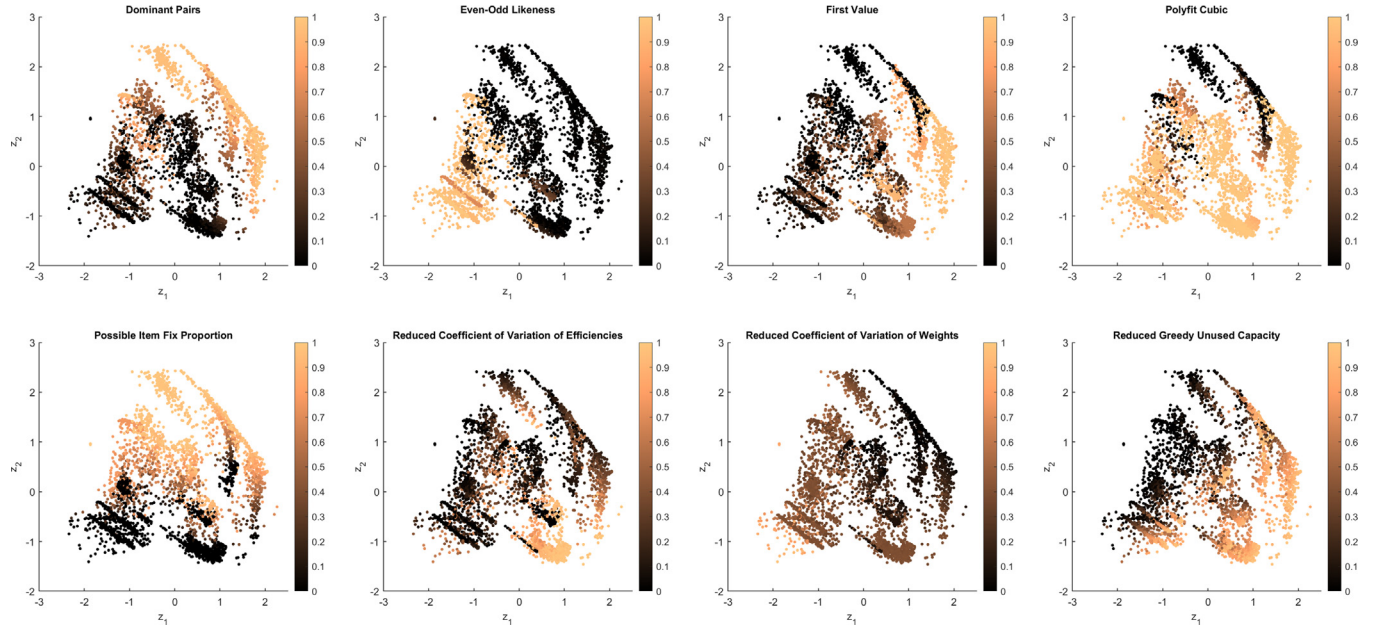

Fig. 5. Distribution of features across initial 0–1KP instance space.

The automated algorithm selection recommendation is then based on examining, for each instance, the predictions of each the three SVMs. In the case of multiple algorithms predicted to achieve good performance, the SVM with the highest precision is trusted to give the recommendation.

The combined SVM selects COMBO as the algorithm most likely to perform the best across most of the instance space, with the exception of a small area at the top on which MINKNAP is selected. Comparing the individual algorithm predictions we see that MINKNAP is predicted to perform well across about the upper half of the space, but the COMBO prediction has higher precision and

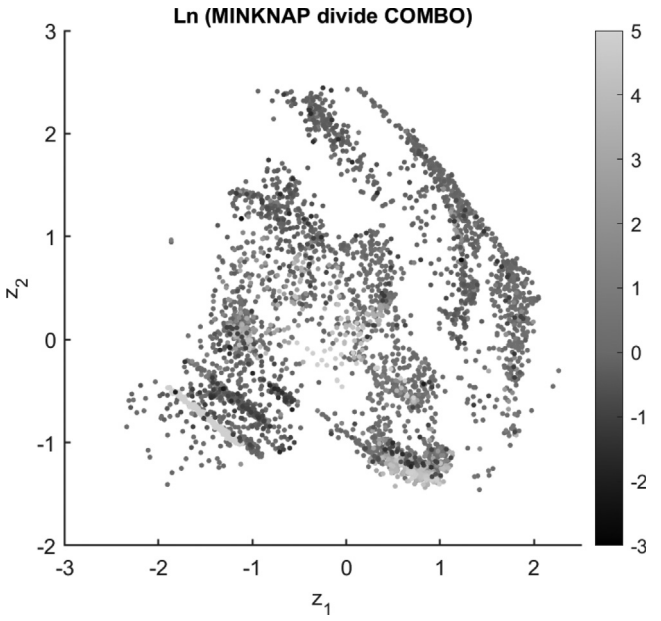

Fig. 6. Comparison between MINKNAP and COMBO performance using the quantity $log_e\left(\frac{t_{\text{MINKNAP}}}{t_{\text{COMBO}}}\right)$. Very dark points indicate instances where MINKNAP is significantly faster than COMBO. Grey points indicate similar performance, with paler points indicating faster COMBO times. The color range is truncated above 5; for the few instances beyond this point COMBO is clearly the superior algorithm..

hence overrules it. This outcome matches with our overall understanding of these algorithms; the MINKNAP algorithm can only be expected to outperform COMBO when applied to easier instances where the more sophisticated and complicated elements of COMBO are unnecessary. The SVM does not select EXPKNAP anywhere in the instance space.

Fig. 8 shows which algorithms actually have good performance for each instance, and the best algorithm for each instance. The predictions made by the SVM appear reasonable when compared to this figure. Furthermore, even though the SVM only considers whether an individual algorithm's performance is good or bad (according to our definition) without any further detail, the SVM prediction also seems reasonable with respect to the more nuanced view of the performance data provided by Figs. 4 and 6.

4. Generating new instances

The existing classes of knapsack instances available in the literature cover a wide range of problem instance characteristics, but do not necessarily provide a representative sample of all possible knapsack instances. Attempting to create a fully representative set of test instances is likely to be both infeasible in terms of the number of instances required, and arguably unnecessary given the set of possible knapsack instances may include many which are unlikely to be encountered in practical applications.

It is nevertheless useful to consider how the set of existing instance classes can be extended to obtain new instances which have novel properties and are potentially difficult to solve. In particular, we expect that the instance space will give us a more reliable and illuminating perspective with respect to the Knapsack Problem as a whole if we repeat the Instance Space Analysis with a more diverse and broadly representative set of test instances.

Given the initial set of existing knapsack instances and their resulting instance space presented in the previous sections, we now seek to produce new instances which are dissimilar to any existing instances. In general this means we are interested in filling 'holes' in the instance space or pushing beyond the outer boundary of the existing instances in directions which appear to correspond to more difficult instances. As such we set two categories of target points. The points in the interior of the instance space are manually placed in areas where our existing instance subset is sparse. The points on the exterior of the instance space are placed on the boundary of the instance space calculated by the toolkit, as follows: Consider a hyper-cube in $8D$ where each vertex corresponds to a combination of the maximum or minimum known values for each feature. This hyper-cube will loosely enclose all the instances in $\mathbf{I}$. However, some of these vertexes represent combinations of features that are unlikely to occur simultaneously due to observed correlations between features. To prune these unlikely vertices, the correlation between any two features $\mathbf{f}_i$ and $\mathbf{f}_j, r_{i,j}$, is compared. Given a user-defined threshold ρ, a vertex is considered unlikely to contain simultaneously the upper and lower bounds of $\mathbf{f}_i$ and $\mathbf{f}_j$ if $r_{i,j} > \rho$, or the upper (or lower) bounds of $\mathbf{f}_i$ and $\mathbf{f}_j$ if $r_{i,j} < -\rho$. The edges connecting all remaining vertices are then projected into the $2D$ space, whose convex hull now represents the empirical limits of the instance space. We select a subset of these vertices (and points on the lines between these vertices) with particular emphasis on areas of the space which our current data set suggests are more likely to contain hard instances. In all, we select 26 target points across the existing instance space (see Fig. 9 (left)).

Given these target points, we propose several methods for generating new instances which lie near these points in the instance space. While there are many possible methods for creating new knapsack instances with particular characteristics, a trade-off exists between the application of human intuition and brute-force computational effort. We consider several of these methods in the following sections.

4.1. Generating weakly structured instances

One method for generating novel knapsack instances is to apply a genetic algorithm (GA) which directly chooses the problem data for each instance (i.e. some or all of the item profits, item weights and capacity). The large variety of instances which can potentially be produced by this approach is both a strength and a weakness. Any given target point represents a particular combination of instance feature values and, in theory, any somewhat random (unstructured) approach that seeks to alter the data to minimise distance of the projected instance to the target point should be able to produce an instance with the desired features. However, finding such an instance from the huge space of possible instances with no structural constraints is considerably more difficult than when using a more structured method. This approach places greater emphasis on computational effort as opposed to human intuition; however some human input is still required to choose the genetic algorithm representation and parameters so as to obtain the best results.

In principle one could represent a knapsack instance as a vector simply containing its item profits, item weights and capacity in one order or another. However, we found that such a totally unstructured representation is not particularly effective at the task of generating instances near a particular point in the $2D$ instance space. Instead, we opt for the following minimally structured approach.

Each individual in the GA encoding represents a set of 10 potential knapsack instances, each of which has the same items but differing capacity fractions in $0.05, 0.15, \ldots, 0.95$. Each instance has 1000 items, each of which has weight equal to its index i.e. the first item has weight 1, the second item has weight 2, and so on. Each individual is defined by a vector of item profits; the first profit corresponds to the item with weight 1, etc. The fitness of an individual is evaluated by projecting all of its 10 potential instances into the instance space and finding the minimum distance between any of

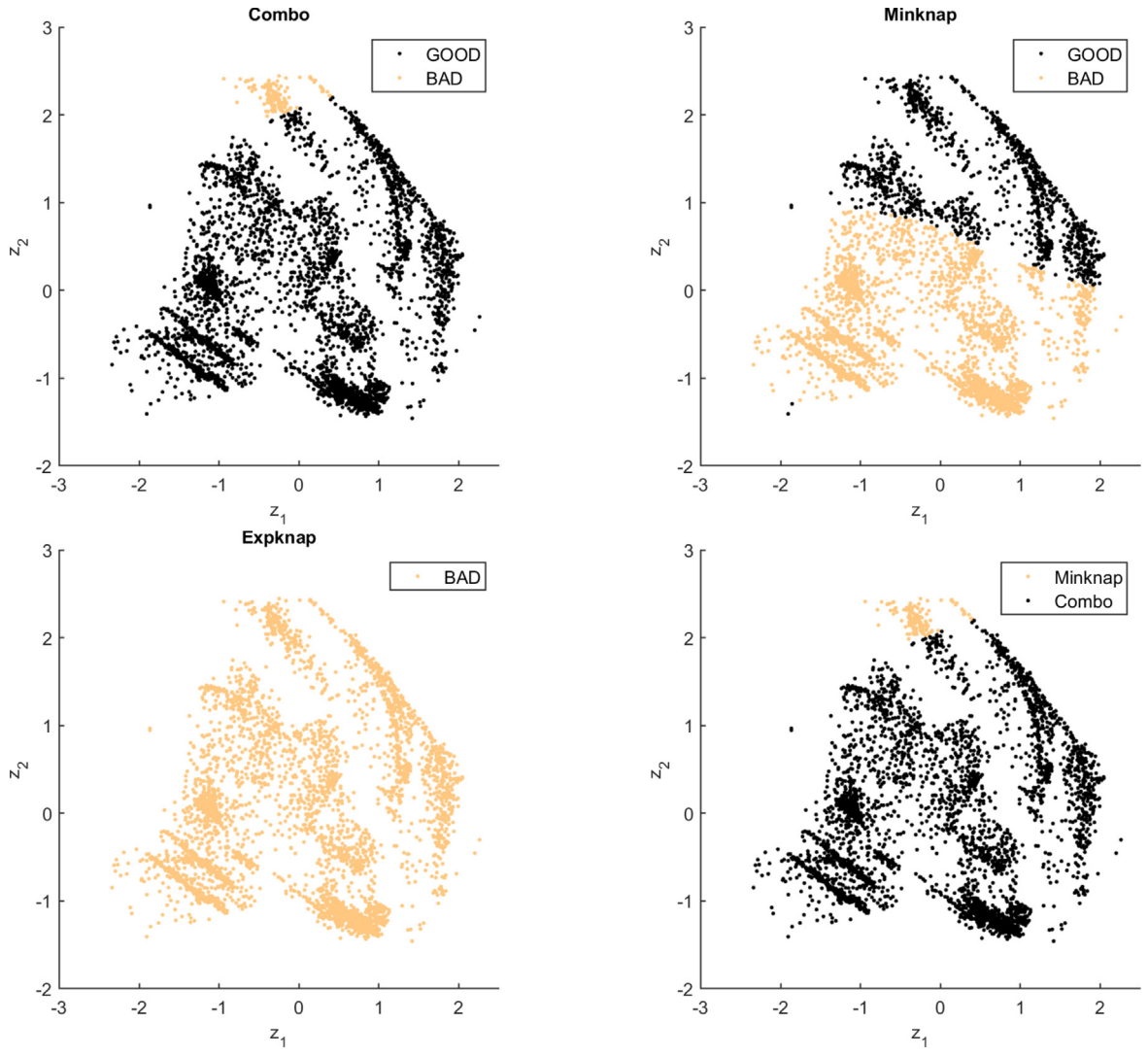

Fig. 7. Individual and combined SVM predictions for algorithm performance.

these instances and the target point; we are attempting to minimise this quantity.

For each target point we create an initial population of 100 individuals by finding the 100 instances from our original instance subset which are closest to the target point, then converting them to a vector of profits by the following procedure. First, for each weight possessed by at least one item in the original instance, we calculate the average profit of the items which have that weight. Second, we perform linear interpolation to assign profits to the remaining weights. Any weights which are lighter or heavier than all items in the original instance are assigned a profit of 1. Finally, we round all profits to the nearest integer.

For each of the 26 target points, at the end of the genetic algorithm procedure we collect the 10 fittest individuals, and from each individual create one new instance using whichever capacity fraction from the set $\{0.05, 0.15, \ldots, 0.95\}$ yields the closest instance to the target point. This results in a total of 260 new instances.

4.2. Generating strongly structured instances

A second approach for generating knapsack instances in the vicinity of a target point in the 2*D* instance space is to design a generation procedure which can produce a variety of instances based on its input parameters, then apply a genetic algorithm to these parameters to find the best setting for each target point. This method leverages both human intuition and computational effort to extend the instance space.

An obvious drawback of this approach is that each individual generation procedure is limited in terms of the types of instance it can produce, and so some regions of the instance space may simply be unreachable by the generation procedure, no matter what parameters are used. By using several different generation procedures, each of which has its own potential range, we can explore a larger proportion of the instance space. In this paper we consider four instance generation procedures, which are summarised in Table 3.

In each case the initial population of 50 individuals is created by applying a random uniform distribution to each parameter independently. Crossover and mutation functions use the default MATLAB genetic algorithm settings. The fitness of an individual is evaluated by creating 50 knapsack instances using the parameters specified by the individual and taking a weighted average of their distances from the target point, weighting the closer instances more strongly than the further instances.

For each of the four generation procedures and 26 target points, at the end of the genetic algorithm procedure we collect the fittest

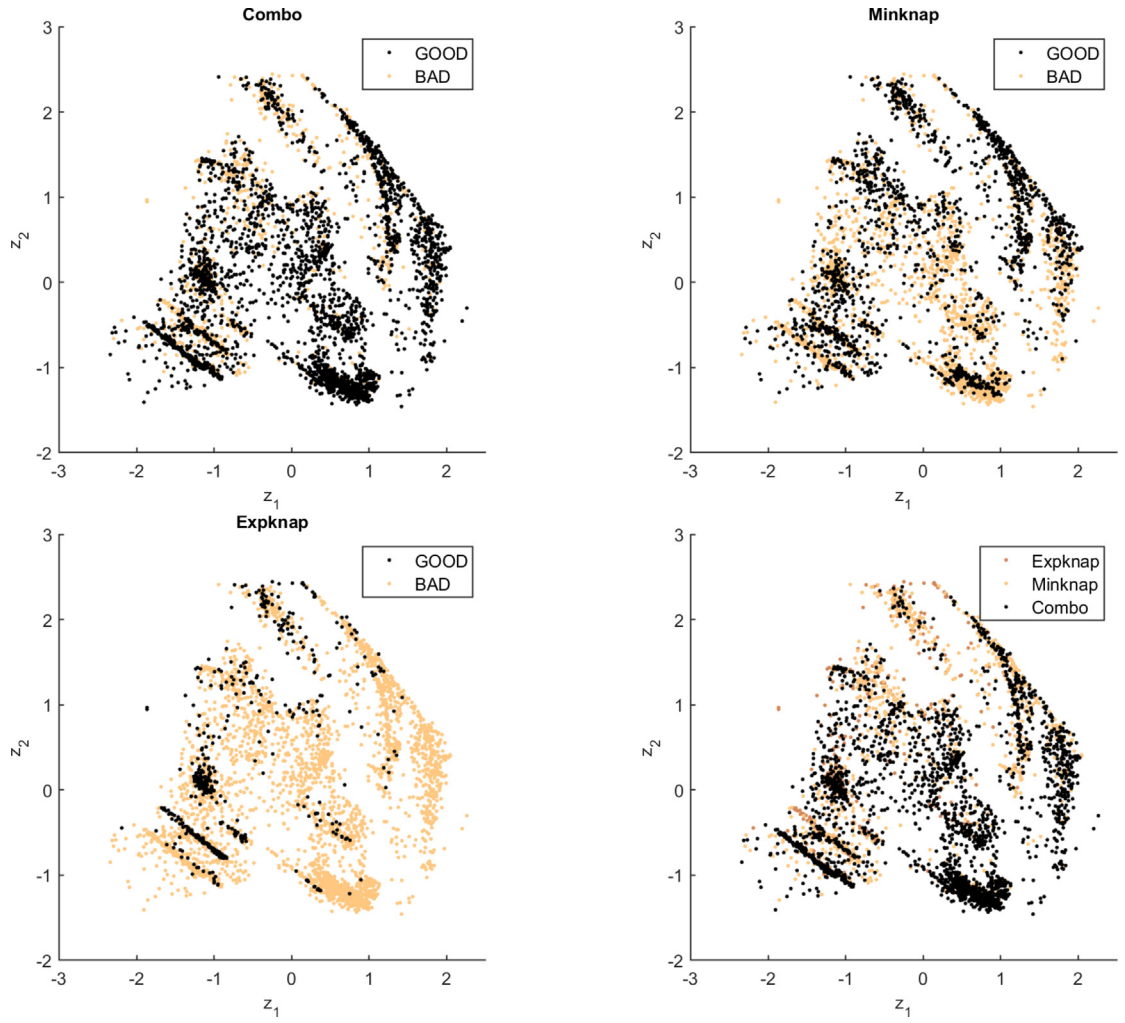

Fig. 8. Actual good/bad performance data for each algorithm (within 20% of fastest algorithm's runtime or not); best algorithm for each instance in lower-right plot.

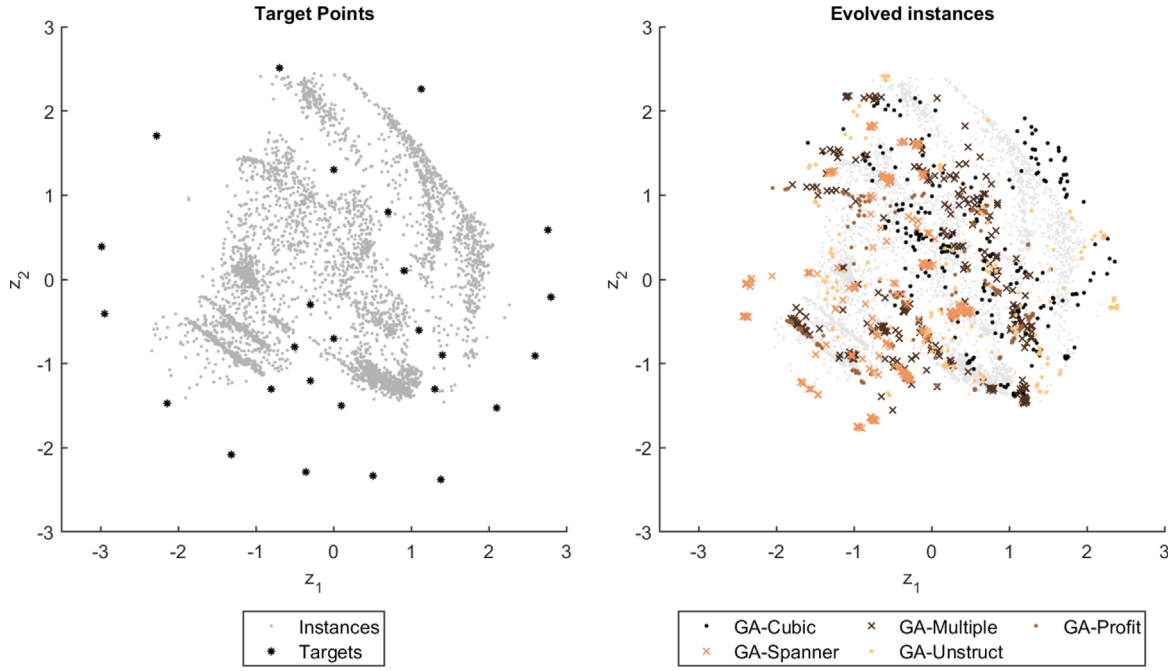

Fig. 9. Target points in the initial instance space for genetic algorithm instance generators (left) and new instances produced by various genetic algorithms (right).

Table 3
Instance generation procedures for use with genetic algorithm parameter tuning to reach target points

Generation Procedure	Parameters and Generation Procedure
Profit Ceiling-like	• knapsack capacity fraction in $[0,1]$ • m, minimum allowed weight/ R in $[0,1]$ • parameter to round to multiplies of in $\{2,3,4,5\}$ (d in Pisinger (2005)) • binary variable defining whether to create a Profit Ceiling or Random Ceiling-type instance • f, fraction of items to have weights such that their weight is equal to their profit, in $[0,1]$ Generate item profits as per Profit Ceiling and Random Ceiling classes, but generate weights with the following procedure. For each item, with probability f assign it a weight greater than m such that its profit will be equal to its weight; choose from any of the acceptable weights with equal probability. With probability $1-f$, assign it a weight greater than m such that its profit will not be equal to its weight in the same way.
Cubic Fit	• knapsack capacity fraction in $[0,1]$ • four values $a_1,\ldots,a_4$ in $[0,1]$ For $j=1,\ldots,4$ set $b_j=(2.88a_j^3-a.32b_j^2+2.44a_j)R$ to shift the curve in favour of profits closer to $R/2$. Find the unique cubic function $f(x)$ which passes through the points $(0,b_1)$, $(R/3,b_2)$, $(2R/3,b_3)$ and (R,b_4). Then generate item weights with a uniform random distribution between 1 and R and assign each item's profit as $p_i=\max\{1,\min\{R,\lceil f(w_i)\rceil\}\}$.
Spanner	• knapsack capacity fraction in $[0,1]$ • maximum multiplier to be applied to the spanner items m in $\{5,6,\ldots,15\}$ • two pairs of parameters (a_1,d_1) and (a_2,d_2), all in $[0,1]$, which determine the weight and profit of the spanner items For $j=1,2$ generate the weight and profit of the two spanner items as follows: $w_j=\left\lceil\frac{d_j}{m}R\cos(\frac{\pi}{2}a_j)\right\rceil$ $p_j=\left\lceil\frac{d_j}{m}R\sin(\frac{\pi}{2}a_j)\right\rceil$ Then generate the spanner instance as in the Spanner-type instance classes.
Multiple Strongly Correlated-like	• knapsack capacity fraction in $[0,1]$ • Three parameters k_1 in $[0,1]$, k_2 in $[0,1]$ and d in $\{3,4,\ldots,10\}$ • f, fraction of items which will have weight divisible by d For each item, with probability f assign it a weight divisible by d (with an equal chance for any acceptable weight) and with probability $1-f$ assign it a weight not divisible by d in the same way. Then assign the profit of each item as in Pisinger (2005): if mod $(w_i,d)=0$ then $p_i=\lfloor\frac{R}{2}k_1\rfloor+w_i$, and $p_i=\lfloor\frac{R}{2}k_2\rfloor+w_i$ otherwise.

individual, and create 10 new instances using the parameters it specifies. This results in a total of 1040 new instances.

4.3. Projecting the generated instances into the initial instance space

Fig. 9 (right) shows the projection of the instances produced by all five genetic algorithm approaches into the original 2*D* instance space. Each of the structured approaches has expanded on the coverage area of the original instance class or classes on which they are based. However, as expected, each of the structured approaches is typically not very good at producing instances which are very different to the underlying instance class.

The Unstructured approach produces new instances in most parts of the instance space which were previously explored, but does not expand the instance space very much besides a few instances on the right side and top-left corner. The Profit Ceiling-based approach also does not expand the space significantly and only produces instances in the lower-left part of the space. The Spanner-based approach produces new instances only on the left-hand side of the space, but notably expands the space around the bottom-left corner. The Cubic-based approach produces instances in the middle and on the right-hand side of the space, expanding the space a little on the right; the Multiple Strongly Correlated-like approach does much the same on the left-hand side of the space.

Having generated new instances which expand the boundaries and fill holes in our original instance space, we may now calculate a new and improved feature subset and instance space projection based on a wider and more representative instance subset.

5. Updating the instance space

As mentioned in Section 2, the ISA methodology is *iterative*, i.e., an initial instance space is created and explored based on currently available meta-data, as shown in Section 3. Gaps in the instance space can then be examined to assess regions where new instances can increase the diversity and coverage of the instance space, as was indicated in Section 4. Now, with these new instances added to $\mathscr{I}$, a different set of features may best describe the algorithms' performance and the 2*D* axes best defining the instance space are likely to change. Therefore, we perform a second Instance Space Analysis with the updated meta-data. The projection equation which produces the new 2*D* instance space is as follows:

$$\begin{bmatrix} Z_1 \\ Z_2 \end{bmatrix} = \begin{bmatrix} 0.4253 & 0.5372 \\ -0.0800 & -0.5138 \\ -0.5251 & 0.2202 \\ 0.3891 & 0.2391 \\ -0.1634 & 0.0404 \\ -0.1539 & 0.2850 \\ -0.3959 & -0.4090 \\ -0.3117 & -0.2180 \\ -0.2269 & 0.2652 \\ -0.3708 & 0.2672 \end{bmatrix}^{\mathrm{T}} \begin{bmatrix} \text{Dominant Pairs} \\ \text{Correlation Coeff.} \\ \text{Approximation Gap} \\ \text{Possible Fix Prop.} \\ \text{First Weight} \\ \text{First Profit} \\ \text{Polyfit Quadratic} \\ \text{Even} - \text{Odd Likeness} \\ \text{Greedy Unused Capacity} \\ \text{Red. Coeff. Var. Effic} \end{bmatrix} \tag{3}$$

Of the features included in this projection matrix, the Dominant Pairs, Possible Item Fix Proportion, Even–Odd Likeness, reduced-knapsack Coefficient of Variation of Efficiencies, and First Profit were also included in the first projection. This lends further weight to our observations in the previous section that the first four of these features were strongly associated with instance difficulty. The Greedy Unused Capacity feature is also used in both projections, although in the first projection it was applied to the reduced knapsack. The Polyfit Quadratic feature in this projection acts similarly to the Polyfit Cubic feature in the first projection. This leaves the Correlation Coefficient, Approximation Gap and First Weight features as additions to the new projection.

5.1. Where are the hard knapsack instances?

The distribution of each algorithm's performance metric and distribution of instance classes across the updated instance space are shown in Figs. 10 and 11 respectively. The harder instances are found in the lower left-hand side of the space, in the negative direction with respect to both Z_1 and Z_2. As in the previous projection, the difficult high-correlation instances occupy a dense cluster (in this case around $Z_1=-2$ and $Z_2=0$). The Subset Sum and

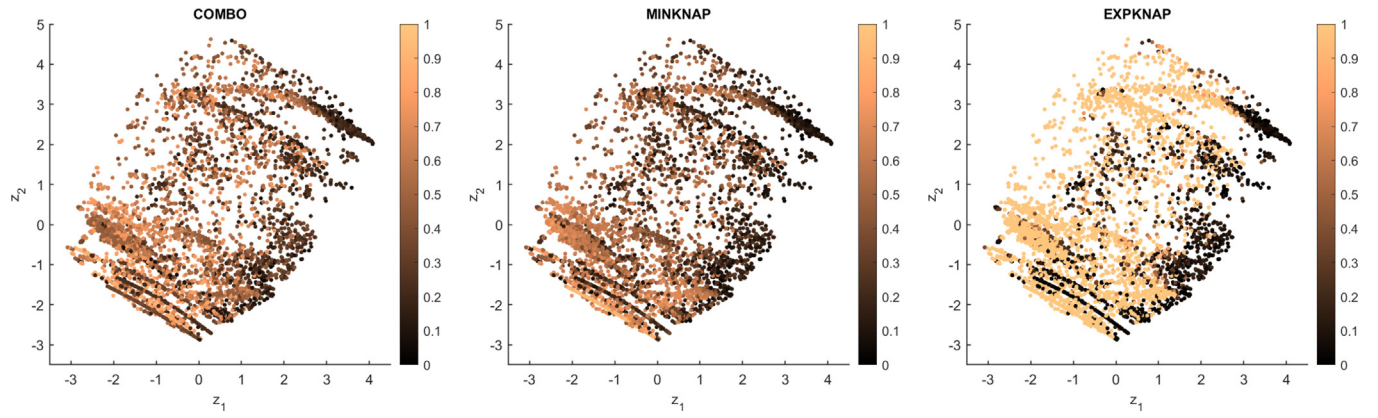

Fig. 10. Algorithm performance across the updated instance space for COMBO (left), MINKNAP (middle), and EXPKNAP (right). The color scale is CPU time, normalized with the fastest time in black and slowest time in orange. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

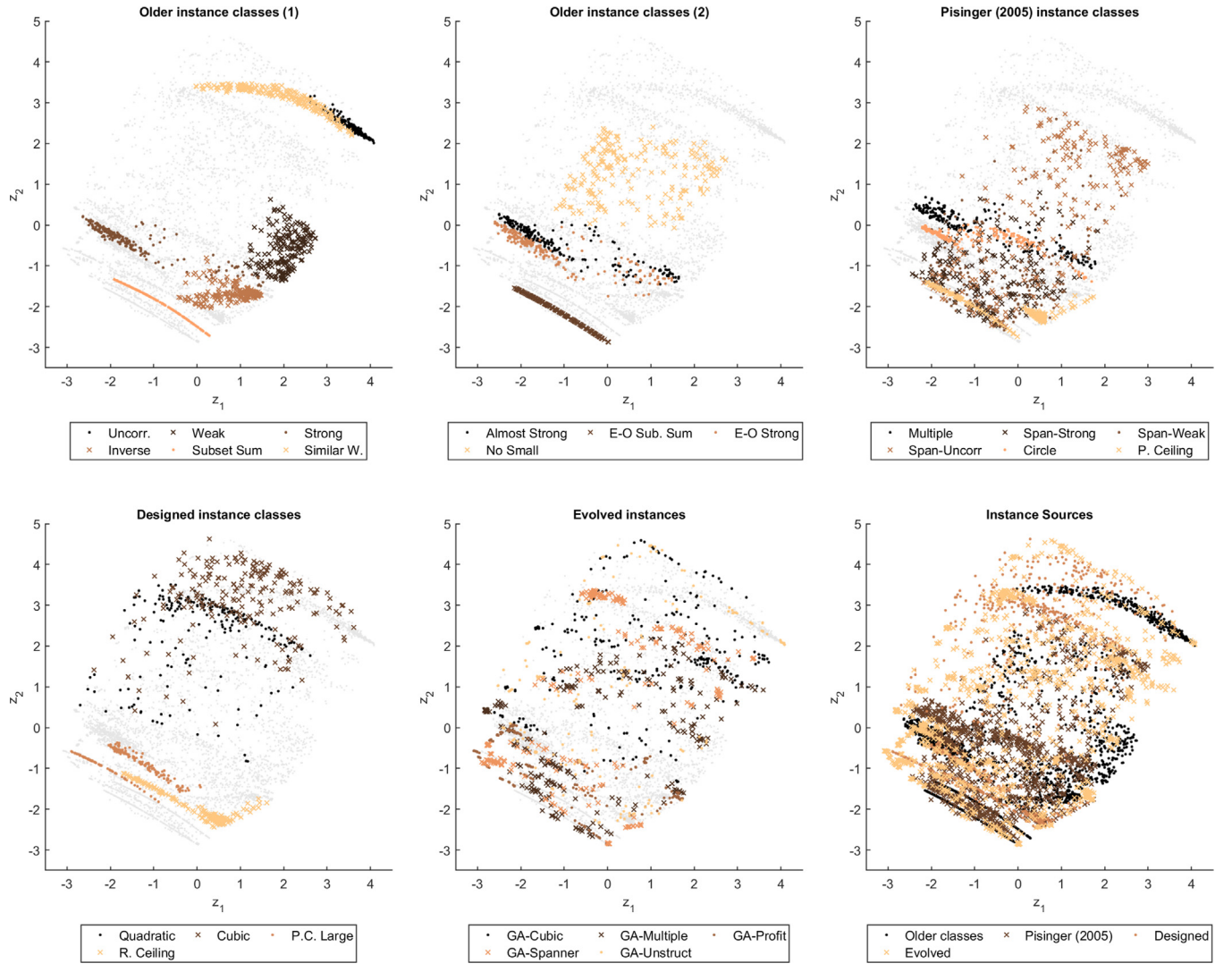

Fig. 11. Instance classes within the updated instance space.

Profit Ceiling instances occupy several lines along the lower-left side of the space (around $Z_1 = -1$ and $Z_2 = -2$). The Spanner instances are spread more widely across the instance space, although the more difficult highly-correlated instances are in close proximity to the Subset Sum and Profit Ceiling lines.

Compared to the previous instance space, the low-density holes in the updated instance space are much smaller, and the Profit Ceiling Large Only instances are considerably closer to the other Profit Ceiling and Subset Sum instances than they are to the high-density cluster. However, the distinction between the low-capacity and

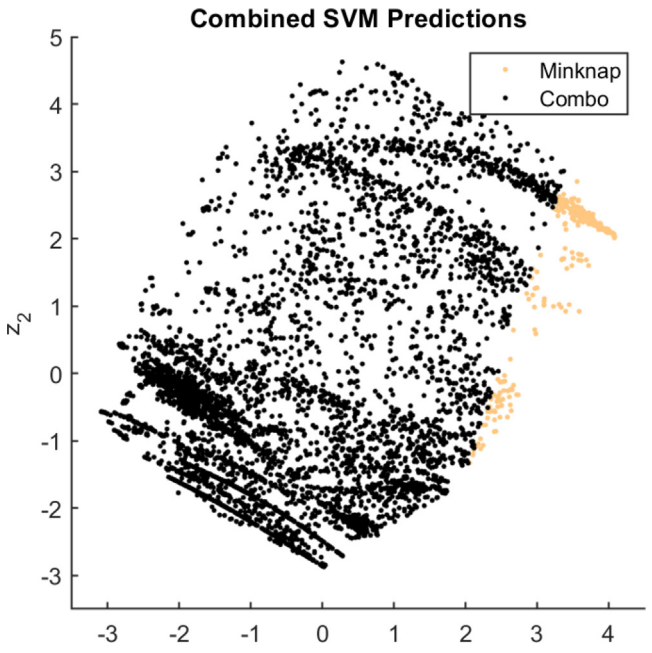

Fig. 12. Algorithm portfolio recommendations using SVM predictions in the updated instance space.

high-capacity Profit Ceiling instances is not as clear cut, and the Multiple and Circle instance classes are more mingled with the other high-correlation instances. While an ideal instance space is difficult to quantify, it is clear that both the original and the updated instance space offer us their own useful insights into 0–1KP.

The general properties of the footprints learned by the support vector machines in the updated 2*D* instance space are similar to those obtained for the initial space. The combined SVM predictions are shown in the form of an automated algorithm selection in Fig. 12 and follow the same pattern as those obtained for the initial instance space; COMBO is predicted to perform best in all areas of the space except a small subset of easy instances on which MINKNAP is predicted to perform best.

6. Conclusions

This paper has provided a visual approach to answer the long-standing question "where are the hard knapsack instances?". Using the recently developed methodology of Instance Space Analysis, we have shown how the strengths and weaknesses of three algorithms for the 0–1KP variant of knapsack problems can be compared objectively across the space of possible test instances, rather than losing valuable information by summarising "on-average" performance. We have demonstrated the areas of strength and weakness for each algorithm, and shown how combining algorithm performance with feature distributions across the instance space can add insights into the conditions under which a given algorithm is expected to perform well or poorly.

An initial instance space revealed opportunities to generate additional test instances to fill gaps and extend the boundaries of the instance space. Updating the instance space with the augmented meta-data created a denser and more comprehensive instance space. The similarities in the selected feature set and overall properties of the two instance spaces suggest that the instance space representation has converged; this gives us greater confidence to say that the insights we have obtained through analysis of these instance spaces are not based on a quirk of our chosen instance subset, but rather give us a greater understanding of 0–1KP as a whole.

Of course, there are always limitations, and opportunities to extend the ideas for future research. One limitation of this study is that the feature set we have proposed includes some features that cannot be evaluated without sorting the items by efficiency, which in many cases takes longer than actually solving the knapsack instance using MINKNAP or COMBO. Therefore, while the 2*D* instance space and SVM results have given us insights into the underlying properties of 0–1KP, they are not very useful for rapidly predicting algorithm performance with respect to a specific unseen instance in real time. It may be possible to find a set of similar features and/or heuristics which are simpler to compute but retain predictive utility. Another potential direction for future research is to adapt and extend the ideas behind these features to more complicated variants of the Knapsack problem.

In this study we have enforced a relatively low bound on the number of items and maximum profit and weight coefficients in each KP instance. This is particularly advantageous for the purposes of generating new instances with genetic algorithms in a reasonable time frame. The downside is that the conclusions drawn in this paper should be treated with some caution if they are applied to a set of instances with substantially larger weights, profits and/or quantity of items. In particular, our instance space contains no representatives from instance classes whose definitions inherently require large weights (such as the Avis knapsack class used in Martello et al. (1999) or the original Similar Uncorrelated Weights class), and so it is likely unsafe to generalise our results to instances derived from these classes.

It would be especially interesting to revisit this analysis in the event that a new algorithm for 0–1KP is devised which is competitive with COMBO in terms of performance when applied to some subset of the more difficult instances, but which has substantially different computational characteristics. We have tested the exact MIP solver CPLEX (version 12.8) as well as another algorithm known as "balknap" (Kellerer et al., 2004), and projected their performance metric into the instance space using MATILDA (see Fig. B.13 in Appendix B). These algorithms only perform well compared with COMBO on a small proportion of our instance subset, and including these algorithms does not substantially change the recommendations of the SVM given in Fig. 12. We hope that by making all of the meta-data publicly available as a library problem on MATILDA (Smith-Miles et al., 2019), other researchers can add new algorithms to the instance space to understand strengths and weaknesses, and support objective comparison with existing algorithms.

CRediT authorship contribution statement

Kate Smith-Miles: Conceptualization, Methodology, Formal analysis, Investigation, Writing - original draft, Supervision, Project administration, Funding acquisition. **Jeffrey Christiansen:** Formal analysis, Software, Investigation, Data curation, Writing - original draft, Visualization. **Mario Andrés Muñoz:** Formal analysis, Methodology, Investigation, Validation, Writing - original draft, Visualization.

Acknowledgements

We are grateful to the two reviewers and editors for their valuable suggestions. Funding was provided by the Australian Research Council through grant FL140100012. The authors are grateful to Samuel Fairchild for his assistance with feature calculations, and Dr. Neelofar for her work on the development of the MATILDA

online tool for Instance Space Analysis available at https://matilda.unimelb.edu.au

Appendix A. Computational results for main algorithms

The computational results for the three main algorithms in this study - COMBO, MINKNAP and EXPKNAP - are presented in Table A.4, for the various classes of instances.

Appendix B. Computational results for additional algorithms

The computational tests used to compare CPLEX and BALKNAP with COMBO were performed on the Spartan HPC system using one physical core (Intel(R) Xeon(R) Gold 6254 CPU @ 3.10 GHz) with 21 GB RAM for each test. Since this is a different CPU type to those used for the tests comparing EXPKNAP, MINKNAP and COMBO in the main body of the paper (as summarised in Table A.4), the runtime results of COMBO summarised in

Table A.4
Runtime data for 0–1KP algorithms with respect to each instance class, generated with $n = 1000$ items and data range parameter $R = 1000$. All times are in milliseconds. Cases where an algorithm failed to terminate are treated as a runtime of 15 s.

	COMBO		MINKNAP		EXPKNAP	
	MEAN(STD)	RANGE	MEAN(STD)	RANGE	MEAN(STD)	RANGE
Uncorrelated	0.03(0.02)	[7.93 × 10^{-3},0.19]	0.03(0.02)	[8.30 × 10^{-3},0.21]	0.11(0.15)	[5.26 × 10^{-3},0.79]
Weakly Correlated	0.06(0.05)	[9.74 × 10^{-3},0.39]	0.05(0.04)	[0.01,0.29]	0.41(0.59)	[4.71 × 10^{-3},4.00]
Strongly Correlated	0.33(0.35)	[8.70 × 10^{-3},2.34]	8.20(8.90)	[0.01,65.22]	1.33e + 04(4.56e + 03)	[5.83 × 10^{-3},1.50e + 04]
Subset Sum	0.04(0.01)	[0.03,0.11]	0.06(0.04)	[0.01,0.19]	0.04(0.06)	[4.83 × 10^{-3},0.51]
Inverse Strongly Correlated	0.43(0.45)	[0.01,3.25]	7.52(8.52)	[0.01,33.47]	1.27e + 04(5.38e + 03)	[0.01,1.50e + 04]
Almost Strongly Correlated	0.26(0.34)	[8.27 × 10^{-3},2.19]	0.69(1.57)	[0.01,11.65]	3.33e + 03(6.08e + 03)	[5.79 × 10^{-3},1.50e + 04]
Similar Uncorrelated Weights	0.92(1.63)	[7.30 × 10^{-3},11.83]	1.38(1.90)	[9.22 × 10^{-3},8.12]	1.06e + 04(6.72e + 03)	[5.27 × 10^{-3},1.50e + 04]
Even–Odd Strongly Correlated	0.44(0.49)	[0.01,2.53]	4.23(4.89)	[0.01,34.96]	1.30e + 04(4.93e + 03)	[7.89 × 10^{-3},1.50e + 04]
Even–Odd Subset Sum	0.07(0.03)	[0.06,0.22]	651.19(443.78)	[15.72,3.39e + 03]	1.50e + 04(0.00)	[1.50e + 04,1.50e + 04]
No Small Weights	0.06(0.04)	[0.01,0.22]	0.07(0.07)	[0.01,0.61]	4.86(15.86)	[4.62 × 10^{-3},206.32]
Spanner Uncorrelated	2.82(3.26)	[0.01,24.70]	3.45(2.87)	[9.53 × 10^{-3},22.40]	1.38e + 04(4.08e + 03)	[4.94 × 10^{-3},1.50e + 04]
Spanner Weakly Correlated	5.28(7.00)	[0.01,52.29]	6.32(12.73)	[8.94 × 10^{-3},156.32]	1.45e + 04(2.76e + 03)	[5.21 × 10^{-3},1.50e + 04]
Spanner Strongly Correlated	7.94(9.75)	[0.02,95.94]	6.65(5.93)	[0.01,55.70]	1.44e + 04(2.95e + 03)	[5.39 × 10^{-3},1.50e + 04]
Multiple Correlated	2.42(3.51)	[7.51 × 10^{-3},24.56]	4.48(5.19)	[9.76 × 10^{-3},23.13]	1.16e + 04(6.14e + 03)	[5.76 × 10^{-3},1.50e + 04]
Profit Ceiling	9.33(17.03)	[8.84 × 10^{-3},68.30]	10.86(20.38)	[0.01,81.85]	9.72e + 03(7.15e + 03)	[3.63 × 10^{-3},1.50e + 04]
Circle	2.77(2.99)	[0.01,12.54]	4.38(4.46)	[0.01,17.36]	1.28e + 04(5.10e + 03)	[7.93 × 10^{-3},1.50e + 04]
Quadratic Fit	0.57(1.06)	[7.01 × 10^{-3},5.78]	0.92(1.79)	[0.01,12.87]	1.29e + 04(5.11e + 03)	[6.10 × 10^{-3},1.50e + 04]
Cubic Fit	0.48(0.78)	[9.13 × 10^{-3},4.78]	0.75(1.35)	[9.38 × 10^{-3},11.01]	1.24e + 04(5.55e + 03)	[5.34 × 10^{-3},1.50e + 04]
Random Ceiling	0.06(0.09)	[8.49 × 10^{-3},0.61]	0.23(0.33)	[0.01,2.33]	0.09(0.15)	[4.46 × 10^{-3},1.02]
Profit Ceiling Large Only	21.16(39.04)	[9.08 × 10^{-3},187.18]	24.68(44.19)	[0.02,177.74]	1.10e + 04(6.60e + 03)	[5.67 × 10^{-3},1.50e + 04]
Evolved Cubic Fit	0.56(1.27)	[0.01,9.01]	0.86(2.31)	[0.01,17.22]	8.77e + 03(7.21e + 03)	[7.48 × 10^{-3},1.50e + 04]
Evolved Multiple-like	1.31(2.70)	[8.93 × 10^{-3},19.68]	4.24(6.98)	[0.01,51.49]	8.11e + 03(7.41e + 03)	[5.05 × 10^{-3},1.50e + 04]
Evolved Profit Ceiling-like	6.57(29.54)	[0.01,357.61]	12.12(45.13)	[0.01,357.35]	6.76e + 03(7.36e + 03)	[4.39 × 10^{-3},1.50e + 04]
Evolved Spanner	2.15(2.82)	[9.29 × 10^{-3},17.04]	1.90(2.15)	[9.17 × 10^{-3},12.14]	9.12e + 03(7.33e + 03)	[5.17 × 10^{-3},1.50e + 04]
Evolved Unstructured	0.99(2.25)	[8.52 × 10^{-3},18.29]	1.90(4.81)	[0.01,35.91]	5.16e + 03(6.83e + 03)	[5.40 × 10^{-3},1.50e + 04]

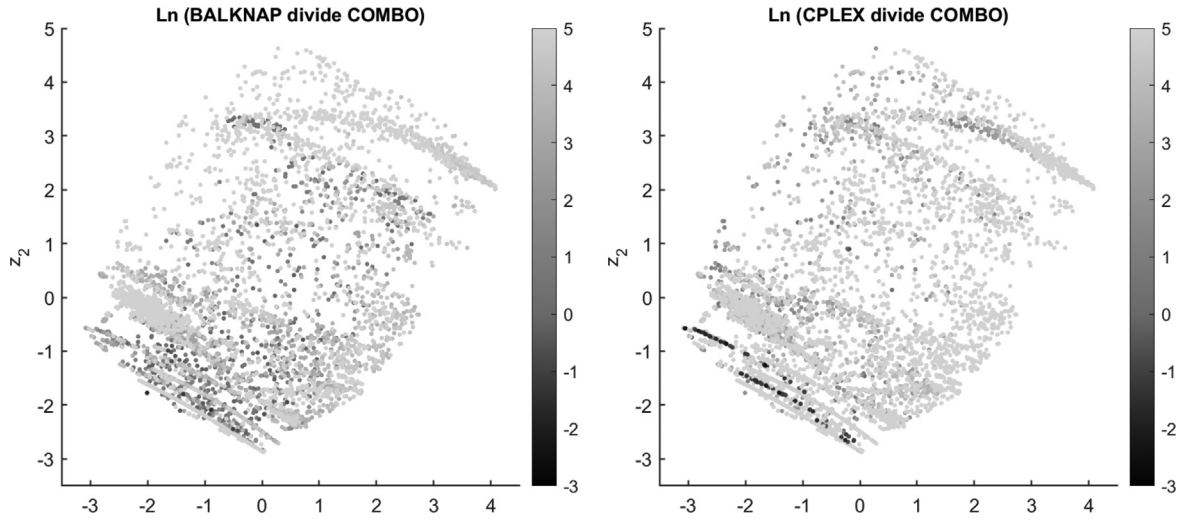

Fig. B.13. Comparison between the runtime performance of Balknap versus COMBO (top), and CPLEX versus COMBO (bottom) in the instance space, using the quantities $log_e\left(\frac{t_{BALKNAP}}{t_{COMBO}}\right)$ and $log_e\left(\frac{t_{CPLEX}}{t_{COMBO}}\right)$ respectively. Very dark points indicate instances where the other algorithm significantly outperforms COMBO. The color range is truncated above 5; for the instances beyond this point COMBO is clearly the superior algorithm. COMBO substantially outperforms BALKNAP and CPLEX across almost all of the instance space.

Table B.5
Runtime data for 0–1KP algorithms with respect to each instance class. All times are in milliseconds. Cases where an algorithm failed to terminate are treated as a runtime of 15 s.

	COMBO		BALKNAP		CPLEX	
	MEAN(STD)	RANGE	MEAN(STD)	RANGE	MEAN(STD)	RANGE
Uncorrelated	0.02(0.01)	[6.64 × 10^{-3},0.08]	1.58(0.80)	[0.46,5.45]	20.10(12.61)	[3.46,65.98]
Weakly Correlated	0.04(0.03)	[7.09 × 10^{-3},0.13]	3.21(2.83)	[0.64,19.91]	32.09(16.07)	[2.72,80.21]
Strongly Correlated	0.22(0.25)	[6.01 × 10^{-3},1.69]	76.42(71.38)	[0.71,268.96]	17.63(25.04)	[4.50,196.20]
Subset Sum	0.03(7.86e-04)	[0.02,0.03]	1.28(1.03)	[0.30,6.20]	8.99(5.70)	[3.78,59.38]
Inverse Strongly Correlated	0.29(0.30)	[9.15 × 10^{-3},1.33]	73.31(80.23)	[1.00,277.70]	17.67(24.26)	[4.25,177.48]
Almost Strongly Correlated	0.17(0.18)	[6.15 × 10^{-3},1.41]	912.55(2.17e + 03)	[0.85,1.35e + 04]	197.40(1.10e + 03)	[5.38,1.50e + 04]
Similar Uncorrelated Weights	0.62(0.81)	[6.03 × 10^{-3},4.24]	5.31e + 03(6.14e + 03)	[0.82,1.50e + 04]	700.67(2.95e + 03)	[3.29,1.50e + 04]
Even–Odd Strongly Correlated	0.32(0.38)	[8.07 × 10^{-3},1.93]	37.82(35.83)	[0.83,133.74]	3.93e + 03(6.51e + 03)	[4.07,1.50e + 04]
Even–Odd Subset Sum	0.06(6.50 × 10^{-3})	[0.05,0.12]	248.40(7.64)	[195.96,259.31]	8.50(3.33)	[3.62,19.27]
No Small Weights	0.05(0.04)	[6.67 × 10^{-3},0.17]	29.80(51.15)	[0.81,425.55]	127.56(1.06e + 03)	[4.28,1.50e + 04]
Spanner Uncorrelated	1.71(1.51)	[0.01,8.08]	3.30(2.48)	[0.50,23.73]	4.28e + 03(6.71e + 03)	[2.77,1.50e + 04]
Spanner Weakly Correlated	3.62(5.00)	[9.71 × 10^{-3},43.39]	4.87(5.41)	[0.37,61.62]	7.18e + 03(7.45e + 03)	[2.53,1.50e + 04]
Spanner Strongly Correlated	3.58(4.75)	[9.85 × 10^{-3},51.45]	6.15(11.03)	[0.34,106.16]	9.07e + 03(7.29e + 03)	[2.23,1.50e + 04]
Multiple Correlated	1.69(2.18)	[5.90 × 10^{-3},11.10]	55.15(59.00)	[0.73,222.01]	5.21e + 03(7.08e + 03)	[2.52,1.50e + 04]
Profit Ceiling	6.71(12.69)	[6.03 × 10^{-3},48.79]	44.01(53.30)	[0.38,222.81]	11.56(7.24)	[4.11,83.23]
Circle	2.16(2.28)	[7.58 × 10^{-3},9.23]	476.35(518.56)	[0.83,2.01e + 03]	12.09(14.95)	[4.56,109.00]
Quadratic Fit	0.42(0.76)	[5.55 × 10^{-3},4.69]	1.19e + 03(1.98e + 03)	[0.82,1.39e + 04]	270.11(1.85e + 03)	[3.60,1.50e + 04]
Cubic Fit	0.38(0.60)	[6.67 × 10^{-3},3.72]	1.47e + 03(2.51e + 03)	[0.93,1.33e + 04]	567.46(2.67e + 03)	[3.78,1.50e + 04]
Random Ceiling	0.04(0.07)	[6.36 × 10^{-3},0.51]	1.92(2.44)	[0.38,26.62]	23.57(16.79)	[4.40,127.30]
Profit Ceiling Large Only	11.55(21.32)	[6.21 × 10^{-3},110.42]	65.09(67.08)	[0.40,238.58]	26.67(27.10)	[4.10,185.97]
Evolved Cubic Fit	0.26(0.57)	[5.84 × 10^{-3},3.48]	505.53(1.31e + 03)	[0.90,9.12e + 03]	1.42e + 03(4.11e + 03)	[3.41,1.50e + 04]
Evolved Multiple-like	0.65(1.37)	[4.80 × 10^{-3},12.10]	4.82e + 03(6.99e + 03)	[0.36,1.50e + 04]	2.33e + 03(5.35e + 03)	[2.79,1.50e + 04]
Evolved Profit Ceiling-like	2.91(11.97)	[4.51 × 10^{-3},123.61]	2.23e + 03(5.09e + 03)	[0.29,1.50e + 04]	194.92(1.49e + 03)	[2.55,1.50e + 04]
Evolved Spanner	0.98(1.22)	[7.33 × 10^{-3},6.14]	2.30(1.58)	[0.32,7.92]	2.29e + 03(5.35e + 03)	[2.45,1.50e + 04]
Evolved Unstructured	0.56(1.22)	[6.40 × 10^{-3},9.84]	1.60e + 03(3.23e + 03)	[0.45,1.50e + 04]	219.02(1.60e + 03)	[2.97,1.50e + 04]

Table B.5 are different in magnitude, though similar in terms of instance class difficulty.

References

Amado, L., Barcia, P., 1993. Matroidal relaxations for 0–1 knapsack problems. Oper. Res. Lett. 14 (3), 147–152. https://doi.org/10.1016/0167-6377(93)90026-D. https://doi-org.ezproxy.lib.rmit.edu.au/10.1016/0167-6377(93)90026-D.

Balas, E., Zemel, E., 1980. An algorithm for large zero-one knapsack problems. Oper. Res. 28 (5), 1130–1154. https://doi.org/10.1287/opre.28.5.1130.

Broyden, C.G., 1970. The convergence of a class of double-rank minimization Algorithms 1. general considerations. IMA J. Appl. Math. 6 (1), 76–90. https://doi.org/10.1093/imamat/6.1.76.

Chung, C.-S., Hung, M.S., Rom, W.O., 1988. A hard knapsack problem. Naval Res. Logist. (NRL) 35 (1), 85–98. https://doi.org/10.1002/1520-6750(198802)35:1<85::AID-NAV3220350108>3.0.CO;2-D.

Chvátal, V., 1980. Hard knapsack problems. Oper. Res. 28 (6), 1402–1411. https://doi.org/10.1287/opre.28.6.1402.

Hall, N.G., Posner, M.E., 2007. Performance prediction and preselection for optimization and heuristic solution procedures. Oper. Res. 55 (4), 703–716. https://doi.org/10.1287/opre.1070.0398.

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. Springer, pp. 73–101.

Hill, R.R., Reilly, C.H., 2000. The effects of coefficient correlation structure in two-dimensional knapsack problems on solution procedure performance. Manage. Sci. 46 (2), 302–317. https://doi.org/10.1287/mnsc.46.2.302.11930.

Hooker, J., 1994. Needed: An empirical science of algorithms. Oper. Res. 42 (2), 201–212.

Hooker, J., 1995. Testing heuristics: We have it all wrong. J. Heuristics 1 (1), 33–42. https://doi.org/10.1007/BF02430364.

Kandanaarachchi, S., Muñoz, M., Hyndman, R., Smith-Miles, K., 2019. On normalization and algorithm selection for unsupervised outlier detection. Data Mining Knowl. Discov. 34, 309–354. https://doi.org/10.1007/s10618-019-00661-z.

Kang, Y., Hyndman, R., Smith-Miles, K., 2017. Visualising forecasting algorithm performance using time series instance spaces. Int. J. Forecast 33 (2), 345–358. https://doi.org/10.1016/j.ijforecast.2016.09.004.

Kellerer, H., Pferschy, U., Pisinger, D., 2004. Exact solution of the knapsack problem. In: Knapsack Problems. Springer, pp. 117–160.

Martello, S., Toth, P., 1988. A new algorithm for the 0–1 knapsack problem. Manage. Sci. 34 (5), 633–644. https://doi.org/10.1287/mnsc.34.5.633.

Martello, S., Toth, P., 1997. Upper bounds and algorithms for hard 0–1 knapsack problems. Oper. Res. 45 (5), 768–778. https://doi.org/10.1287/opre.45.5.768.

Martello, S., Pisinger, D., Toth, P., 1999. Dynamic programming and strong bounds for the 0–1 knapsack problem. Manage. Sci. 45 (3), 414–424. https://doi.org/10.1287/mnsc.45.3.414.

McGeoch, C.C., 2002. Experimental analysis of algorithms. In: Handbook of Global Optimization, Springer, pp. 489–513..

B. Meade, L. Lafayette, G. Sauter, D. Tosello, Spartan HPC-Cloud Hybrid: Delivering Performance and Flexibility (4 2017). doi:10.4225/49/58ead90dceaaa..

Muñoz, M., Smith-Miles, K., 2017. Performance analysis of continuous black-box optimization algorithms via footprints in instance space. Evol. Comput. 25 (4), 529–554. https://doi.org/10.1162/EVCO_a_00194.

Muñoz, M., Smith-Miles, K., 2020. Generating new space-filling test instances for continuous black-box optimization. Evol. Comput. 28 (3), 379–404. https://doi.org/10.1162/evco_a_00262.

Muñoz, M., Villanova, L., Baatar, D., Smith-Miles, K., 2018. Instance spaces for machine learning classification. Mach. Learn. 107 (1), 109–147. https://doi.org/10.1007/s10994-017-5629-5.

Muñoz, M., Smith-Miles, K., 2020. Instance space analysis: A toolkit for the assessment of algorithmic power, Source code is available at https://github.com/andremun/InstanceSpace..

Pferschy, U., Pisinger, D., Woeginger, G.J., 1997. Simple but efficient approaches for the collapsing knapsack problem. Discrete Appl. Math. 77 (3), 271–280. https://doi.org/10.1016/S0166-218X(96)00134-5.

Pisinger, D., 1995. An expanding-core algorithm for the exact 0–1 knapsack problem. Eur. J. Oper. Res. 87 (1), 175–187. https://doi.org/10.1016/0377-2217(94)00013-3.

Pisinger, D., 1997. A minimal algorithm for the 0–1 knapsack problem. Oper. Res. 45 (5), 758–767. https://doi.org/10.1287/opre.45.5.758.

Pisinger, D., 1999a. Core problems in knapsack algorithms. Oper. Res. 47 (4), 570–575. https://doi.org/10.1287/opre.47.4.570.

Pisinger, D., 1999b. Core problems in knapsack algorithms. Oper. Res. 47 (4), 570–575. https://doi.org/10.1287/opre.47.4.570.

Pisinger, D., 2005. Where are the hard knapsack problems?. Comput. Oper. Res. 32 (9), 2271–2284. https://doi.org/10.1016/j.cor.2004.03.002.

Pisinger, D., Toth, P. 1998. Knapsack problems. In: Handbook of Combinatorial Optimization, vol. 1, Kluwer Acad. Publ., Boston, MA, pp. 299–428..

Pisinger, D., Saidi, A., 2017. Tolerance analysis for 0–1 knapsack problems. Eur. J. Oper. Res. 258(3) 866–876. www.scopus.com..

Rice, J., 1976. The algorithm selection problem. In: Advances in Computers, vol. 15, Elsevier, pp. 65–118. doi:10.1016/S0065-2458(08)60520-3..

Smith-Miles, K., Bowly, S., 2015. Generating new test instances by evolving in instance space. Comput. Oper. Res. 63, 102–113. https://doi.org/10.1016/j.cor.2015.04.022.

Smith-Miles, K., Lopes, L., 2012. Measuring instance difficulty for combinatorial optimization problems. Comput. Oper. Res. 39 (5), 875–889. https://doi.org/10.1016/j.cor.2011.07.006.

Smith-Miles, K., Baatar, D., Wreford, B., Lewis, R., 2014. Towards objective measures of algorithm performance across instance space. Comput. Oper. Res. 45, 12–24. https://doi.org/10.1016/j.cor.2013.11.015.

Smith-Miles, K., Muñoz, M., Neelofar, 2019. MATILDA: Melbourne algorithm test instance library with data analytics, Available at https://matilda.unimelb.edu.au..

Wolpert, D., Macready, W., 1997. No free lunch theorems for optimization. IEEE Trans. Evol. Comput. 1 (1), 67–82. https://doi.org/10.1109/4235.585893.