A novelty-search approach to filling an instance-space with diverse and discriminatory instances for the knapsack problem

Alejandro Marrero[1], Eduardo Segredo[1], Coromoto León[1] and Emma Hart[2]

[1] Departamento de Ingeniería Informática y de Sistemas, Universidad de La Laguna, San Cristóbal de La Laguna, Tenerife, Spain
{amarrerd, esegredo, cleon}@ull.edu.es
[2] School of Computing, Edinburgh Napier University, UK
e.hart@napier.ac.uk

Abstract. We propose a new approach to generating synthetic instances in the knapsack domain in order to fill an instance-space. The method uses a novelty-search algorithm to search for instances that are diverse with respect to a feature-space but also elicit discriminatory performance from a set of target solvers. We demonstrate that a single run of the algorithm per target solver provides discriminatory instances and broad coverage of the feature-space. Furthermore, the instances also show diversity within the performance-space, despite the fact this is not explicitly evolved for, i.e. for a given 'winning solver', the magnitude of the performance-gap between it and other solvers varies across a wide-range. The method therefore provides a rich instance-space which can be used to analyse algorithm strengths/weaknesses, conduct algorithm-selection or construct a portfolio solver.

Keywords: Instance generation, novelty search, evolutionary algorithm, knapsack problem, optimisation

1 Introduction

The term *instance-space*—first coined by Smith-Miles *et al* [16]—refers to a high-dimensional space that summarises a set of instances according to a vector containing a list of measures features derived from the instance-data. Projecting the feature-vector into a lower-dimensional space (ideally 2D) enables the instance-space to be visualised. Solver-performance can be superimposed on the visualisation to reveal regions of the instance-space in which a potential solver outperforms other candidate solvers. The low-dimensional visualisation of the instance-space can then be used in multiple ways: to understand areas of the space in which solvers are strong/weak, to perform algorithm-selection or to assemble a portfolio of solvers.

The ability to generate a useful instance-space however depends on the availability of a large set of instances. These instance sets should ideally (1) cover a

high proportion of the 2D-space, i.e. instances are diverse with respect to the *features* defining the 2D-space; (2) contain instances on which the portfolio of solvers of interest exhibit discriminatory performance; (3) contain instances that highlight diversity in the *performance-space* (i.e. highlight a range of values for the performance-gap between the winning solver and the next best solver), in order to gain further insight into the relative performance of different solvers.

On the one hand, previous research has focused on evolving new instances that are maximally discriminative with respect to solvers [1,3,13] (i.e. maximise the performance-gap between a target and other solvers), but tend not to have explicit mechanisms for creating instances that are diverse w.r.t feature-space. On the other hand, space-filling approaches [14] directly attempt to fill gaps in the feature-space, but tend not to account for discriminatory behaviour. The main contribution of our work is therefore in proposing an approach based on Novelty Search (NS) [9] that is simultaneously capable of generating a set of instances which are diverse with respect to a feature space *and* exhibit discriminatory but diverse performance with respect to a portfolio of solvers (where diversity in this case refers to variation in the magnitude of the performance gap). The latter results from forcing the search to explore areas of the feature-space in which one solver outperforms others only by a small amount, which would be overlooked by methods that attempt to optimise this. Furthermore, only one run of the method is required to generate the instance set targeted to each particular solver considered, where each solution of the NS corresponds to one KP instance.

We evaluate the approach in the Knapsack Problem (KP) domain, using a portfolio of stochastic solvers (Evolutionary Algorithms - EAs), extending previous work on instance generation which has tended to use deterministic portfolios. Finally, we explain in the concluding section why we believe the method can easily be generalised both to other domains and other solvers.

2 Related work

The use of EAs to target generation of a set of instances where one solver outperforms others in a portfolio is relatively common. For instance, in the bin-packing domain, Alissa *et al.* [1] evolve instances that elicit discriminatory performance from a set of four heuristic solvers. Plata *et al.* [13] synthesise discriminatory instances in the knapsack domain, while there are multiple examples of this approach to generate instances for Travelling Salesman Problem (TSP) [3, 15]. All of these methods follow a similar approach: the EA attempts to maximise the performance gap between a target solver and others in the portfolio. Hence, while the methods are successful in discovering instances that *optimise* this gap, depending on the search-landscape (i.e. number and size of basins of local optima), multiple runs can converge to very similar solutions. Furthermore, these methods focus only on discrimination and therefore there is no pressure to explore the 'feature-space' of the domain. An implicit attempt to address this in TSP is described in [6], in which the selection method of the EA is altered to

favour offspring that maintain diversity with respect to a chosen feature, as long as the offspring have a performance gap over a given threshold. Again working in TSP, Bossek *et al.* [3] tackle this issue by proposing novel mutation operators that are designed to provide better exploration of the feature-space however while still optimising for performance-gap. In contrast to the above, Smith-Miles *et al* [14] describe a method for evolving new instances to directly fill gaps in an *instance-space* which is defined on a 2D plane, with each axis representing a feature derived from the instance data. While this targets filling the instance-space, it does not pay attention to whether the generated instances show discriminatory behaviour on a chosen portfolio.

As noted in the previous section, our proposed approach addresses the above issue using a novelty-search algorithm to generate diverse instances that demonstrate statistically superior performance for a specified target algorithm compared to the other solvers in the portfolio.

3 Novelty Search for Instance Generation: Motivation

NS was first introduced by Lehman *et al* [9] as an attempt to mitigate the problem of finding optimal solution in deceptive landscapes, with a focus on the control problems. The core idea replaces the objective function in a standard evolutionary search process with a function that rewards novelty rather than a performance-based fitness value to force exploration of the search-space. A 'pure' novelty-search algorithm rewards only novelty: in the case of knapsack instances, this can be defined w.r.t a set of user-defined features describing the instance. However, as we wish to generate instances that are both diverse but also illuminate the region in which a single solver outperforms others in a portfolio, we use a modified form of NS in which the objective function reflects a weighted balance between diversity and performance, where the latter term quantifies the performance difference between a target algorithm and the others in the portfolio.

Given a descriptor x, i.e., typically a multi-dimensional vector capturing features of a solution, the most common approach to quantify novelty of an individual is via the *sparseness* metric which measures the average distance between the individual's descriptor and its k-nearest neighbours. The main motivation behind the usage of descriptors is to obtain a deeper representation of solutions via their features. These features are problem dependent.

The k nearest-neighbours are determined by comparing an individual's descriptor to the descriptors of all other members of the current population and to those stored in an external *archive* of past individuals whose descriptors were highly novel when they originated. Sparseness s is then defined as:

$$s(x) = \frac{1}{k} \sum_{i=0}^{k} dist(x, \mu_i) \tag{1}$$

where μ_i is the ith-nearest neighbour of x with respect to a user-defined distance metric *dist*.

The archive is supplemented at each generation in two ways. Firstly, a sample of individuals from the current population is randomly added to the archive with a probability of 1% following common practice in the literature [17]. Secondly, any individual from the current generation with sparseness greater than a predefined threshold t_a is also added to the archive.

In addition to the archive described above which is used to calculate the sparseness metric that drives evolution, a separate list of individuals (denoted as the solution set) is incrementally built as the algorithm runs: this constitutes the final set of instances returned when the algorithm terminates, again following the method of [17]. At the end of each generation, each member of the current population is scored against the solution set by finding the distance to the nearest neighbour ($k = 1$) in the solution set. Those individuals that score above a particular threshold t_{ss} are added to the solution set. The solution set forms the output of the algorithm.

It is important to note that the solution set does not influence the sparseness metric driving the evolutionary process: instead, this approach ensures that each solution returned has a descriptor that differs by at least the given threshold t_{ss} from the others in the final collection. Finally, both the archive and the solution set grow randomly on each generation depending on the diversity discovered without any limit in their final size.

4 Methods

We apply the approach to generating instances for the KP, a commonly studied combinatorial optimisation problem with many practical applications. The KP requires the selection of a subset of items from a larger set of N items, each with profit p and weight w in such a way that the total profit is maximised while respecting a constraint that the weight remains under the knapsack capacity C. The main motivation behind choosing the KP over other optimisation problems is the lack of literature about discriminatory instance generation for this problem in contrast to other well-known NP-hard problems such as the TSP.

4.1 Instance Representation and Novelty Descriptor

A knapsack instance is described by an array of integer numbers of size $N \times 2$, where N is the dimension (number of items) of the instance of the KP we want to create, with the weights and profits of the items stored at the even and odd positions of the array, respectively. The capacity C of the knapsack is determined for each new individual generated as 80% of the total sum of weights, as using a fixed capacity would tend to create insolvable instances. From each instance, we extract a set of features to form a vector that defines the descriptor used in the sparseness calculation shown in Equation 1. The features chosen are shown below, i.e. the descriptor is a 8-dimensional vector taken from [13] containing: *capacity of the knapsack; minimum weight/profit; maximum weight/profit; average item efficiency; mean distribution of values between profits and weights*

Table 1: Parameter settings for EA_{solver}. The crossover rate is the distinguishing feature for each configuration.

Parameter	Value
Population size	32
Max. Evaluations	1e5
Mutation rate	1 / N
Crossover rate	0.7, 0.8, 0.9, 1.0
Crossover	Uniform Crossover
Mutation	Uniform One Mutation
Selection	Binary Tournament Selection

($N \times 2$ integer values representing the instance); standard deviation of values between profits and weights. We evolve fixed size instances containing $N = 50$ items, hence each individual describing an instance contains 100 values describing pairs of (profit, weight). In addition, upper and lower bounds were set to delimit the maximum and minimum values of both profits and weights.[3] All algorithms were written in C++.[4]

4.2 Algorithm Portfolio

While in principle the portfolio can contain any number or type of solvers, we restrict experiments to a portfolio containing four differently configured versions of an EA. Parameter tuning can significantly impact EA performance on an instance [12]. That is the reason why we are interested in addressing the generation of instances for specific EA configurations rather than different heuristics or algorithmic schemes. As a result, it is expected that different configurations of the same approach cover different regions of the instance space. Each EA (EA_{solver}) is a standard generational with elitism GA [10] with parameters defined in Table 1. The four solvers differ only in the setting of the crossover rate, i.e. $\in$ 0.7, 0.8, 0.9, 1.0, which are common values used in the literature.

4.3 Novelty Search Algorithm

The NS approach ($EA_{instance}$), described by Algorithm 1, evolves a population of instances: one run of the algorithm evolves a diverse set of instances that are tailored to a chosen target algorithm. All parameters are given in Table 2. We note that, since $EA_{instance}$ is time-consuming, its population size, as well as its number of evaluations, were set by trying to get a suitable trade-off between the results obtained and the time invested for attaining them.

To calculate the fitness of an *instance* in the population (Algorithm 2), two quantities are required: (1) the novelty score measuring the sparseness of the

[3] The description of an instance follows the general method of [13], except that they converted the real-valued profits/weights to a binary representation.

[4] The source code, instances generated and results obtained are available in a GitHub repository: `https://github.com/PAL-ULL/ns_kp_generation`.

Table 2: Parameter settings for $EA_{instance}$ which evolves the diverse population of instances. This approach was executed 10 times for each targeted algorithm for statistical purposes.

Parameter	**Value**
Knapsack items (N)	50
Weight and profit upper bound	1,000
Weight and profit lower bound	1
Population size	10
Crossover rate	0.8
Mutation rate	1 / ($N \times 2$)
Evaluations	2,500, 5,000, 10,000, 15,000
Repetitions (R)	10
Distance metric	Euclidean Distance
Neighbourhood size (k)	3
Thresholds (t_a, t_{ss})	3.0

instance and (2) the performance score measuring difference in average performance over R repetitions between the target algorithm and the best of the remaining algorithms. The *novelty* score s (sparseness) for an instance is calculated according to Equation 1 using the descriptor x detailed in Section 4.1, and the Euclidean distance between the vectors as the *dist* function. The *performance* score ps is calculated according to Equation 2, i.e., the difference between the mean profit achieved in R repetitions by the target algorithm and the maximum of the mean profits achieved in R repetitions by the remaining approaches of the portfolio (where profit is the sum of the profits of items included in a knapsack). The reader should consider that, in order to generate discriminatory instances for different algorithms, the target algorithm must vary from one execution to another. In other words, our approach does not generate biased instances for different algorithms in one single execution.

$$ps = \text{target\_mean\_profit} - max(\text{other\_mean\_profit}) \tag{2}$$

Finally, the fitness f used to drive the evolutionary process is calculated as a linearly weighted combination of the novelty score s and the performance score ps of an instance, where ϕ is the performance/novelty balance weighting factor.

$$f = \phi * ps + (1 - \phi) * s \tag{3}$$

5 Experiments and results

Experiments address the following questions:

1. What influence does the number of generations have on the distribution of evolved instances?

Algorithm 1: Novelty Search

Input: N, k, $MaxEvals$, $portfolio$
initialise(*population*, N);
evaluate(*population*, $portfolio$);
archive = $\emptyset$;
feature_list = $\emptyset$;
for $i = 0$ *to* $MaxEvals$ **do**
  parents = select(*population*);
  offspring = reproduce(*parents*);
  offspring = evaluate($offspring$, $portfolio$, $archive$, k) (Algorithm 2);
  population = update(*population*, $offspring$);
  archive = update_archive(*population*, $archive$);
  solution_set = update_ss(*population*, *solution_set*);
end
return *solution_set*

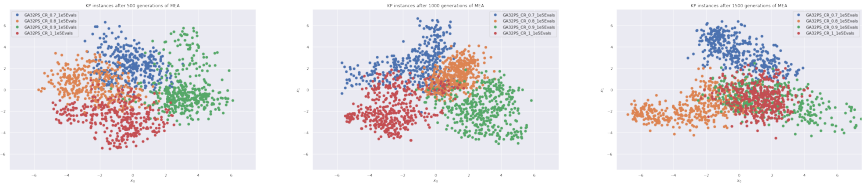
Algorithm 2: Evaluation method

Input: $offspring$, $portfolio$, $archive$, k
for *instance in* $offspring$ **do**
  for *algorithm in* $portfolio$ **do**
    apply *algorithm* to solve *instance* R times;
    calculate mean profit of *algorithm*
  end
  calculate the novelty score($offspring$, archive, k) (Equation 1);
  calculate the performance score($offspring$) (Equation 2);
  calculate fitness($offspring$) (Equation 3);
end
return $offspring$

2. To what extent do the evolved instances provide diverse coverage of the instance space?
3. What effect does the parameter ϕ that governs the balance between novelty and performance have on the diversity of the evolved instances?
4. How diverse are the instances evolved for each target with respect to the performance difference between the target algorithm and the best of the other algorithms, i.e. according to Equation 2?
5. Given a set of instances evolved to be tailored to a specific target algorithm, to what extent is the performance of the target on the set statistically significant compared to the other algorithms in the portfolio?

5.1 Influence of generation parameter

$EA_{instance}$ was run for 250, 500, 1,000 and 1,500 generations (2,500, 5,000, 10,000 and 15,000 evaluations, respectively). $EA_{instance}$ was run 10 times for each of the four target algorithms and the instances for each run per target were combined.



(a) 500 generations (b) 1000 generations (c) 1500 generations

Fig. 1: Instance representation in a 2D space after applying PCA. Colours reflect the 'winning' algorithm for an instance: red (crossover rate 1.0); green (0.9), orange (0.8), blue (0.7). For more detail about this and following figures please refer to the GitHub repository previously mentioned.

The parameter ϕ describing the performance/novelty balance was set to 0.6. Principal Component Analysis (PCA) was then applied to the feature-descriptor detailed in Section 4.1 to reduce each instance to two dimensions. The results are shown in Figure 1.

From a qualitative perspective, the most separated clusters are seen for the cases of 500 and 1,000 generations. More overlap is observed when running for too few or too many generations. A plausible explanation often noted in the novelty search literature, e.g. [4], is that if the novelty procedure is run for too long, it eventually becomes difficult to locate novel solutions as the search reaches the boundaries of the feasible space. As a result, the algorithm tends to fill in gaps in the space already explored. On the other hand, considering 250 generations does not allow sufficient time for the algorithm to discover solutions that are both novel and high performing. That is the reason why those results are not shown in Figure 1. In the remaining experiments, we fix the generations at 1,000.

In order to quantitatively evaluate the extent to which the evolved instances cover the instance space, we calculate the exploration uniformity (U) metric, previously proposed in [7, 8]. This enables a comparison of the distribution of solutions in the space with a hypothetical Uniform Distribution (UD). First, the environment is divided into a grid of 25 x 25 cells, after which the number of solutions in each cell is counted. Next, the Jensen-Shannon divergence (JSD) [5] is used to compare the distance of the distribution of solutions with the ideal UD. The U metric is then calculated according to Equation 4. The higher the U metric score, the more uniformly distributed the instances and better covered the instance space in a given region. Obtaining a score of 1 proves a perfect uniformity distributed set of solutions.

$$U(\delta) = 1 - JSD(P_\delta, UD) \tag{4}$$

In Equation 4, δ denotes a *descriptor* associated with a solution. Following common practice in the literature and to simplify the computations, this descriptor is defined as the two principal components of each solution extracted after applying PCA to the feature-based descriptor described in Section 4.1.

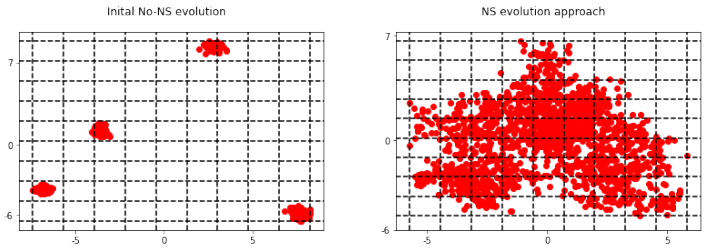
Table 3: Average number of instances generated after running $EA_{instance}$ 10 times for each target algorithm, average coverage metric (U) per run, and total number of unique instances obtained from combining the instances over multiple runs with its corresponding coverage metric (U). No duplicated instances were found when comparing the individual's descriptors.

Target	**Avg. instances**	**Avg. U**	**Tot. instances**	**Tot. U**
GA_0.7	33.4	0.404	334	0.673
GA_0.8	37.6	0.416	376	0.678
GA_0.9	33.9	0.405	339	0.647
GA_1.0	38.3	0.410	383	0.647

Table 3 summarises the average number of instances generated per run and the average coverage metric U per each target algorithm, as well as the total number of unique instances generated and its corresponding coverage metric U per each target approach. Since the portfolio approaches only differ in the crossover rate, we use the term GA_cr with $cr \in \{0.7, 0.8, 0.9, 1.0\}$ to refer to each target algorithm. Considering the instance space, it can be observed that the method is robust in terms of the number of instances generated, as well as in terms of the corresponding U metric values. Values are similar regardless of the particular target approach for which instances were generated.

5.2 Instance space coverage

The left-hand side of Figure 2 shows results from running $EA_{instance}$ with the performance/novelty weighting factor ϕ set to 1, i.e., the EA only attempts to maximise the performance gap and ignores novelty, i.e. an equivalent experiment to that described in [13]. The target algorithm in this case has crossover rate = 0.7. Small groups of clustered instances are observed, with a U value of 0.3772. In contrast, the right-hand side (again with target crossover=0.7) demonstrates that running $EA_{instance}$ with a performance/novelty weighting factor ϕ set to



(a) Evolution without NS (b) Evolution including NS

Fig. 2: Instance representation in a 2D search space after applying PCA comparing two methods of instances generation.

0.6 clearly results in a large coverage of the space with a corresponding U value of 0.7880.

5.3 Influence of the balance between novelty and performance

Recall that the evolutionary process in $EA_{instance}$ is guided by Equation 3, where parameter ϕ balances the contribution of performance and novelty to calculate an individual's fitness. Low values favour feature-diversity, while high values large performance-gaps. We test eight different weighting settings, showing three examples with $\phi \in \{1, 0.7, 0.3\}$ in Figure 3. The target approach was GA_0.7. Results show that a reasonable compromise is obtained with a performance/novelty balance weighting factor ϕ equal to 0.7: instances are clustered according to the target algorithm while maintaining diversity. As ϕ reduces to favour novelty, as expected, coverage increases at the expense of clustered instances.

(a) $\phi = 1.0$ (b) $\phi = 0.7$ (c) $\phi = 0.3$

Fig. 3: Instance representation in a 2D search space after applying PCA for three examples of performance/novelty balance weighting factors used to calculate fitness.

5.4 Comparison of target algorithm performance on evolved instances

The goal of the approach presented is to evolve a diverse set of instances whose performance is tailored to favour a specific target algorithm. Due to the stochastic nature of the solvers, we conduct a rigorous statistical evaluation to determine whether the results obtained on the set of instances evolved for a target algorithm show statistically significant differences compared to applying each of the other algorithms to the same set of instances (Table 4). First, a *Shapiro-Wilk test* was performed to check whether the values of the results followed a normal (Gaussian) distribution. If so, the *Levene test* checked for the homogeneity of the variances. If the samples had equal variances, an ANOVA *test* was done; if not, a *Welch test* was performed. For non-Gaussian distributions, the non-parametric *Kruskal-Wallis* test was used [11]. For every test, a significance level $\alpha = 0.05$ was considered. The comparison was carried out considering the mean profits

Table 4: Statistical analysis. A **win** (↑) indicates significance difference between two configurations and that the mean performance value of the target was higher. A **draw** (↔) indicates no significance difference between both configurations. The number of instances generated for each target approach was 90 (GA_0.7), 101 (GA_0.8), 110 (GA_0.9) and 80 (GA_1.0).

	GA_0.7	**GA_0.8**	**GA_0.9**	**GA_1.0**
GA_0.7		↑ 87 ↔ 3	↑ 69 ↔ 21	↑ 25 ↔ 65
GA_0.8	↑ 100 ↔ 1		↑ 77 ↔ 24	↑ 21 ↔ 80
GA_0.9	↑ 107 ↔ 3	↑ 87 ↔ 23		↑ 18 ↔ 92
GA_1.0	↑ 21 ↔ 59	↑ 61 ↔ 19	↑ 76 ↔ 4	

achieved by each approach at the end of 10 independent executions for each instance generated.

For each target approach A in the first column, the number of 'wins' (↑) and 'draws' (↔) of each target algorithm with respect to other approach B is shown. A 'win' means that approach A provides statistically better performance in comparison to approach B, according to the procedure described above, when solving a particular instance. A 'draw' indicates no significant difference. For example, GA_0.7 provides statistically better performance than GA_0.8 in 87 out of the 90 instances generated for the former. Note that in no case did the target algorithm lose on an instance to another algorithm.

For the three algorithms with crossover rates {0.7, 0.8, 0.9} then for the vast majority of instances, the target algorithm outperforms the other algorithms. However for these three algorithms, it appears harder to find diverse instances where the respective algorithm outperforms the algorithm with configuration 1.0. Thus the results provide some insights into the relative strengths and weaknesses of each algorithm in terms of the size of their footprint within the space (approximated by the number of generated instances).

5.5 Performance Diversity

Finally, we provide further insight into the diversity of the evolved instances with respect to the *performance* space (see Figures 4 and 5). That is, we consider instances that are 'won' by a target algorithm and consider the spread in the magnitude of the performance gap as defined in Equation 2. We note that the approach is able to generate diverse instances in terms of this metric: while a significant number of instances have a relatively small gap (as seen, for instance, by the left skew to the distribution in Figure 4), we also find instances spread across the range (see Figure 5). The instances therefore exhibit performance diversity as well as diversity in terms of coverage of the instance space.

6 Conclusions and further research

The paper proposed an NS-based algorithm to generate sets of instances tailored to work well with a specific solver that are diverse with respect to a feature-space

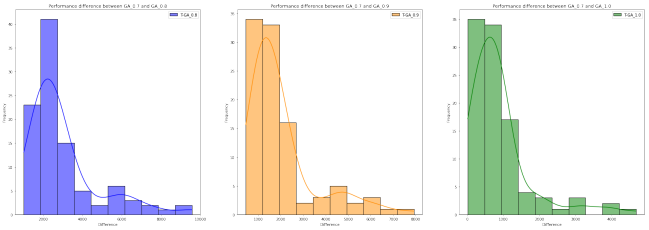


Fig. 4: Histogram showing the distribution of performance gap between the approach GA_0.7 and the remaining approaches by considering the instances generated for the former.

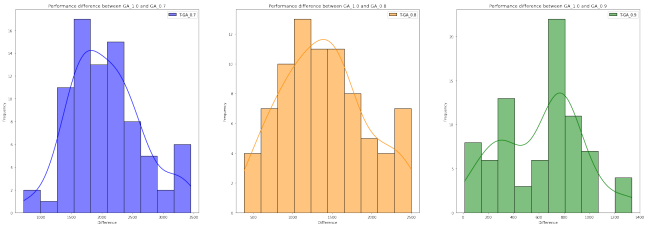


Fig. 5: Histogram showing the distribution of performance gap between the approach GA_1.0 and the remaining approaches by considering the instances generated for the former.

and also diverse with respect to the magnitude of the performance-gap between the target solvers and others in a portfolio.

The results demonstrate that the NS-based method provides larges sets of instances that are considerably more diverse in a feature-space in comparison to those generated by an evolutionary method that purely focuses on maximising the performance gap (i.e. following the method of [13]). It also provides instances that demonstrate diversity in the performance space (Figures 4 and 5). A major advantage of the proposed method is that a single run returns a *set* of diverse instances per target algorithm, in contrast to previous literature for instance generation [1, 13, 15]) that requires repeated runs due to EA convergence, with no guarantee that repeated runs will deliver unique solutions.

The results also shed new insights into the strengths and weaknesses of the four algorithms used, in terms of the size of their footprint in the instance-space, while also emphasising the benefits of algorithm-configuration. Despite only changing one parameter (crossover rate) per configuration, we are able to generate a large set of instances per configuration that are specifically tailored to that configuration, demonstrating that even small changes in parameter values can lead to different performance.

Although our results are restricted to evolving knapsack instances in conjunction with a portfolio of EA-based approaches for generating solutions, we suggest that the the method is generalisable. The underlying core of the approach is an

EA to evolve new instances: this has already been demonstrated to be feasible in multiple other domains, e.g. binpacking and TSP [1, 3]. Secondly, it requires the definition of a feature-vector: again the literature describes numerous potential features relevant to a range of combinatorial domains[5]. At the same time, a basic version of NS was recently used by Alissa *et al.* [2] to evolve instances that are diverse in the *performance-space* for 1D bin-packing, suggesting that other descriptors and other domains are plausible.

Finally, regarding the KP domain, it would be interesting to add the capacity C of the knapsack as a feature of the instances being evolved.

Acknowledgement

Funding from Universidad de La Laguna and the Spanish Ministerio de Ciencia, Innovación y Universidades is acknowledged [2022_0000580]. Alejandro Marrero was funded by the Canary Islands Government "Agencia Canaria de Investigación Innovación y Sociedad de la Información - ACIISI" [TESIS2020010005] and by HPC-EUROPA3 (INFRAIA-2016-1-730897), with the support of the EC Research Innovation Action under the H2020 Programme. This work used the ARCHER2 UK National Supercomputing Service (https://www.archer2.ac.uk).

References

1. Alissa, M., Sim, K., Hart, E.: Algorithm Selection Using Deep Learning Without Feature Extraction. In Genetic and Evolutionary Computation Conference (GECCO '19), July 13–17, 2019, Prague, Czech Re- public. ACM, New York, NY, USA (2019), https://doi.org/https://doi.org/10.1145/332170710.1145/3321707, https://doi.org/10.1145/3321707.
2. Alissa, M., Sim, K., Hart, E.: Automated algorithm selection: from feature-based to feature-free approaches (2022), https://doi.org/https://doi.org/10.48550/ARXIV.2203.1339210.48550/ARXIV.2203.13392, https://arxiv.org/abs/2203.13392
3. Bossek, J., Kerschke, P., Neumann, A., Wagner, M., Neumann, F., Trautmann, H.: Evolving diverse tsp instances by means of novel and creative mutation operators. In: Proceedings of the 15th ACM/SIGEVO Conference on Foundations of Genetic Algorithms. pp. 58–71 (2019)
4. Doncieux, S., Paolo, G., Laflaquière, A., Coninx, A.: Novelty search makes evolvability inevitable. In: Proceedings of the 2020 Genetic and Evolutionary Computation Conference. p. 85–93. GECCO '20, Association for Computing Machinery, New York, NY, USA (2020), https://doi.org/https://doi.org/10.1145/3377930.338984010.1145/3377930.3389840, https://doi.org/10.1145/3377930.3389840
5. Fuglede, B., Topsoe, F.: Jensen-shannon divergence and hilbert space embedding. In: International Symposium onInformation Theory, 2004. ISIT 2004. Proceedings. pp. 31– (2004),

[5] Features for creating instance-spaces in a broad range described in detail from MATILDA https://matilda.unimelb.edu.au/matilda/

https://doi.org/https://doi.org/10.1109/ISIT.2004.136506710.1109/ISIT.2004.1365067

6. Gao, W., Nallaperuma, S., Neumann, F.: Feature-based diversity optimization for problem instance classification. In: International Conference on Parallel Problem Solving from Nature. pp. 869–879. Springer (2016)
7. Gomes, J., Mariano, P., Christensen, A.L.: Devising effective novelty search algorithms: A comprehensive empirical study. GECCO 2015 - Proceedings of the 2015 Genetic and Evolutionary Computation Conference pp. 943–950 (2015), https://doi.org/https://doi.org/10.1145/2739480.275473610.1145/2739480.2754736
8. Le Goff, L.K., Hart, E., Coninx, A., Doncieux, S.: On Pros and Cons of Evolving Topologies with Novelty Search. The 2020 Conference on Artificial Life pp. 423–431 (2020)
9. Lehman, J., Stanley, K.O.: Abandoning objectives: Evolution through the search for novelty alone. Evolutionary Computation **19**(2), 189–222 (2011)
10. Marrero, A., Segredo, E., Leon, C.: A parallel genetic algorithm to speed up the resolution of the algorithm selection problem. In: Proceedings of the Genetic and Evolutionary Computation Conference Companion. p. 1978–1981. GECCO '21, Association for Computing Machinery, New York, NY, USA (2021), https://doi.org/https://doi.org/10.1145/3449726.346316010.1145/3449726.3463160, https://doi.org/10.1145/3449726.3463160
11. Marrero, A., Segredo, E., León, C., Segura, C.: A Memetic Decomposition-Based Multi-Objective Evolutionary Algorithm Applied to a Constrained Menu Planning Problem. Mathematics **8**(11) (2020), https://doi.org/https://doi.org/10.3390/math811196010.3390/math8111960, https://www.mdpi.com/2227-7390/8/11/1960
12. Nannen, V., Smit, S.K., Eiben, A.: Costs and Benefits of Tuning Parameters of Evolutionary Algorithms. Proceedings of the 10th international conference on Parallel Problem Solving from Nature (2008)
13. Plata-González, L.F., Amaya, I., Ortiz-Bayliss, J.C., Conant-Pablos, S.E., Terashima-Marín, H., Coello Coello, C.A.: Evolutionary-based tailoring of synthetic instances for the Knapsack problem. Soft Computing **23**(23), 12711–12728 (2019), https://doi.org/10.1007/s00500-019-03822-w
14. Smith-Miles, K., Bowly, S.: Generating new test instances by evolving in instance space. Computers and Operations Research **63**, 102–113 (2015), https://doi.org/https://doi.org/https://doi.org/10.1016/j.cor.2015.04.022https://doi.org/10.1016/j.cor.2015.04.022, https://www.sciencedirect.com/science/article/pii/S0305054815001136
15. Smith-Miles, K., van Hemert, J., Lim, X.Y.: Understanding tsp difficulty by learning from evolved instances. In: International Conference on Learning and Intelligent Optimization. pp. 266–280. Springer (2010)
16. Smith-Miles, K.A.: Cross-disciplinary perspectives on meta-learning for algorithm selection. ACM Comput. Surv. **41**(1), 1–25 (2009), https://doi.org/https://doi.org/10.1145/1456650.145665610.1145/1456650.1456656, http://doi.acm.org/10.1145/1456650.1456656
17. Szerlip, P.A., Morse, G., Pugh, J.K., Stanley, K.O.: Unsupervised Feature Learning through Divergent Discriminative Feature Accumulation. AAAI'15: Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence (jun 2014), http://arxiv.org/abs/1406.1833