Contents lists available at ScienceDirect

SoftwareX

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

Original software publication

DIGNEA: A tool to generate diverse and discriminatory instance suites for optimisation domains

Alejandro Marrero [a,*], Eduardo Segredo [a], Coromoto León [a], Emma Hart [b]

[a] *Departamento de Ingeniería Informática y de Sistemas, Universidad de La Laguna, San Cristóbal de La Laguna, Spain*
[b] *School of Computing, Edinburgh Napier University, Edinburgh, United Kingdom*

ARTICLE INFO

Article history:
Received 21 November 2022
Received in revised form 22 February 2023
Accepted 6 March 2023

Keywords:
Instance generation
Novelty search
Evolutionary algorithm
Optimisation
Knapsack problem

ABSTRACT

To advance research in the development of optimisation algorithms, it is crucial to have access to large test-beds of diverse and discriminatory instances from a domain that can highlight strengths and weaknesses of different algorithms. The DIGNEA tool enables diverse instance suites to be generated for any domain, that are also discriminatory with respect to a set of solvers of the user choice. Written in C++, and delivered as a repository and as a Docker image, its modular and template-based design enables it to be easily adapted to multiple domains and types of solvers with minimal effort. This paper exemplifies how to generate instances for the Knapsack Problem domain.

Code metadata

Current code version	v1.0.0
Permanent link to code/repository used for this code version	https://github.com/ElsevierSoftwareX/SOFTX-D-22-00386
Code Ocean compute capsule	None
Legal Code License	GPL-3.0 License
Code versioning system used	Git
Software code languages, tools, and services used	C++, OpenMP, CMake, Catch2
Compilation requirements, operating environments & dependencies	C++20 (tested with gcc 10.3.0+), Linux; alternatively Docker.
If available Link to developer documentation/manual	https://dignea.github.io/
Support email for questions	dignea@ull.edu.es

1. Motivation and significance

In any optimisation domain, it is well known that no single solver can best solve all instances, necessitating the use of algorithm-portfolios which collectively provide coverage of the instance space of the problem. Therefore, this leads to the per-instance Algorithm-Selection Problem (ASP) proposed by Rice in 1976 [1], i.e. choose the best solver from a portfolio for a given instance. The ASP has garnered considerable attention over recent years [2] with the raise of Machine Learning approaches to predict either performance of a given algorithm or the label of the best solver using large datasets of instances from the optimisation domain. However, the process of gathering sufficient instances that both cover the feature-space of instances and are discriminatory with respect to the solvers in the portfolio is considerably challenging.

While most of the work in the field has been focused on generating difficult instances in different domains [3–5], recent research has focused on generating instances which are maximally discriminative with respect to a portfolio of solvers proposed for a specific domain, for instance, maximising the performance-gap between a target and other solvers for the Bin-Packing, Travelling Salesman Problem (TSP) and Knapsack Problem (KP) domains, respectively [6–9]. Nevertheless, most of these approaches do not include explicit mechanisms to generate instances that are diverse with respect to the feature-space — they focus only on generating instances that are diverse with respect to solver performance. The work of Smith-Miles does attempts to generate space-filling instances, i.e. in unexplored regions of the feature-space, but only generates ten instances per run and needs to be repeatedly run at each point in the space where an instance is

* Corresponding author.
E-mail addresses: amarrerd@ull.edu.es (Alejandro Marrero), esegredo@ull.edu.es (Eduardo Segredo), cleon@ull.edu.es (Coromoto León), e.hart@napier.ac.uk (Emma Hart).

required. In contrast, alternative proposals attempted to generate instances across domains rely on instances which can be solved by any solver in the portfolio [10], or even on a completely random generation process [11].

Our software, which we refer to as DIGNEA, is a generalisable C++ framework that is capable of either generating instances that are diverse with respect to a feature-space defined by a user, or generating instances that are diverse with respect to a performance vector relating to a pre-defined portfolio, which implicitly also promotes diversity in the feature-space. The term *feature-space* refers to the space defined from the set of features that can be used to define an instance in a optimisation domain. The set of features of an instance is known as its feature-descriptor. On the other hand, generating diverse instances with respect to the performance vector defines the *performance-space* of the portfolio used. The set of performance values with respect to a user pre-defined portfolio is known as its performance-descriptor. Unlike the feature-descriptor, the performance-descriptor of an instance is completely dependent to the algorithms which shape the portfolio.

A run of DIGNEA requires the specification of a target algorithm belonging to the portfolio. A single run generates multiple instances that are diverse with respect to either features or performance and, at the same time, solved better by the chosen target in comparison to any other algorithm in the portfolio. Therefore, a user would typically run DIGNEA once for each target-algorithm in their chosen portfolio.

DIGNEA is based on the usage of Novelty Search (NS) [12], which refers to a type of Evolutionary Algorithm (EA) that rewards novelty, i.e. differences between solutions, rather than the quality of a solution. A pure EA rewards only objective fitness, while a pure NS rewards only novelty, and therefore, tends to result in more exploration. In DIGNEA, as in much other work, the two different types of fitness are combined into a single-objective function via a weighting factor, i.e. when the weight is 1, it becomes a pure EA, while when it is 0, the approach becomes a pure NS.

The software has been used in recent conference work [13] where we first proposed the usage of NS to generate instances in the KP domain for a portfolio consisting of a set of different parameterisations of a traditional EA.

For more detail about the roles of the algorithms in the portfolio and other components, as well as to get further information about the experimental procedure carried out, please, refer to previous work [13].

Finally, it is worth mentioning that the modular structure of DIGNEA could facilitate the work of researchers to create large datasets of instances for any domain, analyse strengths/weaknesses of solvers, conduct algorithm-selection or construct a portfolio of solvers.

2. Software description

DIGNEA is written in modern C++ combining template-based types with creational design patterns that allow users to extend the framework to their needs.

2.1. Software architecture

Interconnections between DIGNEA types, classes and modules are defined by inheritance, a common approach in optimisation software [14]. `AbstractSolver` is the main algorithm interface for defining new algorithms across the entire framework. Moreover, `Problem` is a class which collects the necessary information to define any optimisation problem the user may want to generate instances for. It allows users to define different solution representations via template parameters. To solve a problem, a solution must be defined. `Solution` is a template class which represents a typical solution for an optimisation problem. It includes the variables (genotype) and the objective values (phenotype) of a given solution for a particular problem. Since DIGNEA covers a range of pre-defined solution types, defining your own custom solution type might be optional.

To improve the user experience in DIGNEA, two creational design patterns were considered for its design: *builders and factories*. Builders are used to instantiate algorithms, experiments and Evolutionary Instance Generator (EIG) configurations. At the same time, factories allow variation operators and other components to be created on the fly.

From the previous building blocks, `EvolutionaryInstanceGenerator` (EIG), `AbstractDomain`, `AbstractInstance`, as well as two NS descriptor types, `NSFeatures` and `NSPerformance` were written. `EIG` is the main component of DIGNEA. It implements an NS approach to generate sets of diverse and discriminatory instances for any optimisation problem. `AbstractDomain` defines an instance generation domain for `EIG`. It basically defines a domain to generate instances for an optimisation problem P previously defined in DIGNEA (an object of class `Problem`). `AbstractInstance` is a solution in the `AbstractDomain` domain, i.e. it represents an instance for the optimisation problem P. It includes all the information to construct an actual instance for optimisation problem P.

Fig. 1 shows the relationship among the main components for instance generation in DIGNEA. For example, to instantiate an object of type `EIG`, users must specify the following components: an NS descriptor type, such as `NSFeatures` or `NSPerformance`, created through factory `NSFactory`; a domain to generate instances for (`AbstractDomain`), such as `KPDomain`; the representation of an instance (`AbstractInstance`), such as `KPInstance`; and a portfolio of algorithms for which instances are going to be produced (`AbstractSolver`), such as `EA` or Simulated Annealing (class `SA`).

Classes `AbstractDomain` and `AbstractInstance` are completely dependent since they may include ad-hoc operations, specific attributes and a particular representation.

Therefore, to specify a new domain in DIGNEA, users *must* define at least: an optimisation problem `Problem`, a specific domain `AbstractDomain`, and the representation of an instance of the problem `AbstractInstance`.[1] Colour codes and different types of lines in Fig. 1 reflect the inheritance and extension needs in DIGNEA. Red rectangles with dotted lines represent the base classes that must be extended for specifying new domains. Those in green with dashed lines are the custom types for the KP domain provided with the tool. Finally, yellow rectangles with straight lines are the base types that users do not have to modify necessarily.

Finally, it is worth mentioning at this point that DIGNEA is offered as a Docker image to facilitate the user experience, as well as to make the software portable to different platforms.[2] The image contains all the dependencies and source code of DIGNEA. In order to run the tests or examples, users only need to move inside the `bin` directory and execute the files they want. There is no need to build the software when creating a new container. However, any modifications or additions to the framework will require to rebuild it. Additionally, the building steps are reduced to execute the `build.sh` script that can be found in the root directory.

[1] To see the full documentation and examples of DIGNEA, check the documentation at Github: https://dignea.github.io/.

[2] The Docker image can be accessed through: https://hub.docker.com/r/dignea/dignea.

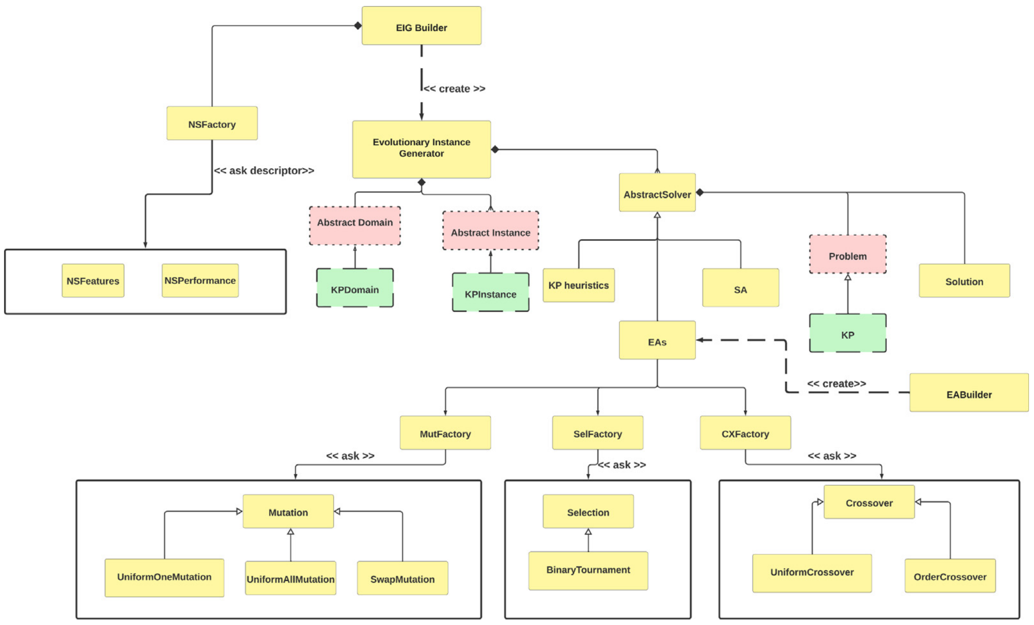


Fig. 1. Relationship of DIGNEA main components for instance generation. Red rectangles with dotted lines represent those classes which must be extended to specify new domains. At the same time, green rectangles with dashed lines refer to the custom types for the KP domain. Yellow rectangles with straight lines are the base types used in DIGNEA that users do not necessarily need to modify. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

2.2. Software functionalities

The principal contribution of DIGNEA is to bring researchers a complete generalisable software to generate instances in any optimisation domain. These instances could be then used for instance space representation, instance characterisation via hand-designed features, automate designed features, algorithm design and selection or even parameter tuning evaluation for algorithms across domains. Moreover, since the generation of a set of instances involves the resolution of those instances with several algorithms, DIGNEA could also be used as an optimisation framework to separately validate the instances generated.

The software is not only extendable from the domain point of view, but also from the portfolio and novelty search descriptors. From the portfolio perspective, even though the software is mainly conceived to use EAs, any other non-EA algorithms could also be included. The current version of DIGNEA includes the following components for instance generation:

- Solvers: Evolutionary Algorithms (`EA` objects), such as First Improve, Generational, Steady-State and Parallel Generational [15]; a Simulated Annealing approach (`SA`), as well as four well-known deterministic heuristics for the KP (`KPHeuristics`).
- Novelty Search descriptor types: Novelty-Search by Features (NSFeatures) and Novelty-Search by Portfolio Performance (NSPerformance).
- Domains: Knapsack Problem (classes `KPDomain`, `KPInstance` and `KP`).
- Builders and Factories: `EIGBuilder`, a builder of `EIG` objects to generate instances; `EABuilder`, a builder of EA configurations; `NSFactory`, an NS descriptor factory; `CXFactory`, a crossover operator factory; `MutFactory`, a mutation operator factory; and a parent selection operator factory `SelFactory`.
- An instance printer class to generate domain dependent instance files using the *insertion operator*.[3] This operator must be defined when creating a new domain through the extension of class `AbstractDomain`.

Regarding the NS descriptors, `NSFeatures` allows to search for diversity in a pre-defined feature-space of the domain. Thus, users must define the set of features which characterise an instance in the domain and how to compute them inside the classes extending `AbstractDomain`. Alternatively, `NSPerformance` searches for diversity in the portfolio-performance space. Here, we search for instances that are diverse with respect to the performance of the solvers without considering any other information. `NSPerformance` is a suitable option for domains where the features are difficult to define or computationally expensive to calculate. We should note at this point that, considering the above NS descriptors, diversity can be calculated using three different distance metrics: Euclidean, Manhattan and Hamming. For further detail, refer to the NS proposal [12].

The evolutionary process performed by EIG is detailed in Fig. 2. Once the specific domain and the portfolio of solvers are configured, the evolutionary process begins. First, a random population of instances are created and evaluated. After that, during G generations, the instances are evolved by following the classical EA scheme. The variation operators (crossover and mutation) are applied before evaluating all the instances with each algorithm in the portfolio. After solving all instances with each algorithm in the portfolio, a fitness value must be assigned to each instance. Here we use the term fitness from the evolutionary computation field to define how suitable an instance is at a specific point during the evolutionary process. Particularly, the fitness is calculated as a linear weighted combination of two values, the performance score and the novelty score. The *performance score ps* of each instance is calculated by using Eq. (1), i.e the difference between the mean performance achieved in R repetitions by the target algorithm, denoted as t_p, when solving the instance at hand; and the maximum of the mean performance achieved in R repetitions by the remaining approaches in the portfolio, defined as o_p, when solving that particular instance. We should note at this point that DIGNEA assumes that the algorithm in the first position of the

[3] C++ documentation: https://en.cppreference.com/w/cpp/io/basic_ostream/operator_ltlt2.

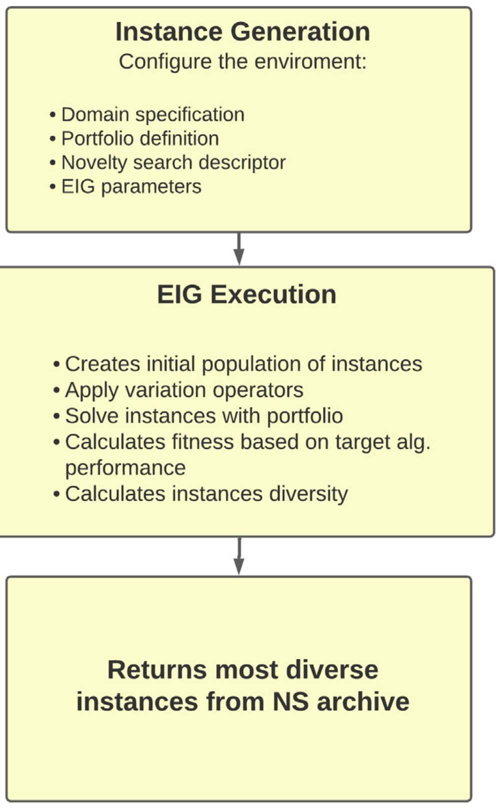

Instance Generation
Configure the enviroment:
• Domain specification
• Portfolio definition
• Novelty search descriptor
• EIG parameters
EIG Execution
• Creates initial population of instances
• Apply variation operators
• Solve instances with portfolio
• Calculates fitness based on target alg. performance
• Calculates instances diversity
Returns most diverse instances from NS archive

Fig. 2. Program flowchart of an instance generation process in DIGNEA.

portfolio is considered the target. Therefore, the user must be careful when specifying the order of the solvers in the portfolio.

$$ps = t_p - \max(o_p) \qquad (1)$$

After that, we calculate the *novelty score* s of each instance in the population depending on the particular descriptor selected. Then, the fitness f is calculated as a linear weighted combination of the novelty score s and the performance score ps, where ϕ is the performance/novelty balance weighting factor (Eq. (2)). Setting low ϕ values will produce more spread and diverse instances, while higher values will generate better grouped and discriminatory instances. Finally, those instances with a novelty score larger than a predefined threshold and a positive performance score are included into an archive to be part of the final set of solutions returned by the approach.

$$f = \phi * ps + (1 - \phi) * s \qquad (2)$$

3. Illustrative examples

In this section, we provide a source code example on how DIGNEA can be used to generate instances in the KP domain. For this, we only need to define the parameters for the domain, the portfolio of solvers and the particular NS descriptor to run the experiment. The documentation available at the Github repository.[4] provides complete tutorials for creating new domains, algorithms and run instance generation experiments.

In our example, we will be using a portfolio of four deterministic KP heuristics [8] to evolve KP instances based on the performance of those solvers. The portfolio is defined as a C++ vector of unique pointers to `AbstractSolver` objects. The order in which the vector is populated is extremely important. The solver in the first position, i.e. that located at position zero in the vector, will be considered as the target solver, and therefore, the instances generated will be tailored so this solver outperforms the remaining solvers in the portfolio when solving those particular instances. Considering the KP domain, we set the number of items in the instances to $N = 50$, and the bounds $w_i, p_i \forall i \in N$ to be in the range $[1, 1000]$. After that, we use `EIGBuilder` to create a unique pointer to an `EIG` object with all the configuration required to generate a set of instances for the KP domain. Notice that `EIGBuilder` facilitates the initialisation process by using the necessary factories, i.e. *withSearch* uses the `NSFactory` type to create a new NS descriptor object of type `NSFeatures` with the remaining arguments. The above is illustrated in the fragment of C++ code shown in Fig. 3. This source code fragment shows that the builder pattern is used to create a unique pointer to a `EIG` object. The methods are self explanatory. Calling the method `run` starts the evolutionary process and the method `getResults` provides the set of instances at the end of the execution.

Fig. 4 shows a Javascript Object Notation (JSON) file generated from the experiment detailed in this section. The JSON files usually contains all the information from the EIG, the domain (`KPDomain` in our example) and a *front* of solutions. The front contains the set of diverse instances generated by the EIG. For each instance, it contains its novelty score, performance score, fitness and the actual instance information. Besides, it may contain its feature values if defined.

Applying a simple data analysis procedure of scalarisation and dimensionality reduction can provide a clear visualisation of the instances produced. Fig. 5 shows the results of two instance generation experiments. The subfigure at the top shows the instances generated by EIG using `NSFeatures` as the NS descriptor, while the subfigure at the bottom shows those instances produced by `NSPerformance`. It can be observed how instances spread across the space but also are divided in different clusters (one per solver in the portfolio), thus facilitating their classification.

4. Impact

Since the definition of the ASP back in 1976, researchers have struggle to address this computationally expensive problem. Traditionally, solving the ASP for a domain involves at least a three step procedure. First, you start by generating and gathering large amount of instances to generate a dataset. Then, after defining a portfolio of solvers, you must undergo a considerable expensive computational experiment, i.e., solving each instance in the dataset with every solver in the portfolio. Thereafter, a selection mechanism must be applied to associate each instance to the solver which obtains the best performance. This process is very time consuming and prone to human mistakes when switching from one step to the next one. Besides, in most cases, the instance generation process was performed randomly and does not guarantee to be representative of the domain or emphasise the strengths/weaknesses of the solvers.

DIGNEA was designed to facilitate this process by combining the previous three steps in one single procedure. The workflow of the software not only ensures that the generated instances are correctly labelled to the best performing solver in the portfolio, but also allow researchers to define the diverse the instances they want to be with respect to each other. Moreover, the modular and template-based architecture of DIGNEA allow researchers to switch domains and include their own solvers in a straightforward manner. Although DIGNEA allows the instance generation process in one single step to be simplified, the above still is a computationally expensive task. However, the software supports

[4] The Github repository of DIGNEA can be accessed through: https://github.com/DIGNEA/dignea.

```
unique_ptr<EIG<IP, IS, KP, OS>> eig =
    EIGBuilder<IP, IS, KP, OS>::create()
        .toSolve(move(instKP))
        .with()
        .weights(fRatio, nRatio)
        .portfolio(algs)
        .evalWith(move(easyEvaluator))
        .repeating(reps)
        .withSearch(NSType::Features,
                move(distance), thresholdNS, k)
        .with()
        .crossover(CXType::Uniform)
        .mutation(MutType::UniformOne)
        .selection(SelType::Binary)
        .withMutRate(mutationRate)
        .withCrossRate(0.8)
        .populationOf(nInstances)
        .runDuring(generations);
eig->run();
auto instances = eig->getResults();
```

Fig. 3. C++ source code fragment to generate KP instances in DIGNEA.

```
{ "algorithm": {
           "portfolio": [
            {  "isTarget": true,  "name": "Default KP"
            },
            {  "isTarget": false, "name": "MPW KP"
            },
            {  "isTarget": false, "name": "MaP KP"
            },
            {  "isTarget": false, "name": "MiW KP"
            }
        ],
    (...) // Other relevant information
    "front": {
        "0": { // First instance
            "novelty": 6938.35498046875,
            "features": {
                "Q": 23678.0,
                "avg_eff": 0.7300000190734863,
                "max_p": 991.0,
                "max_w": 985.0,
                "mean": 538.2100219726563,
                "min_p": 8.0,
                "min_w": 48.0,
                "std": 298.83905029296875
            },
            "fitness": 116.80326843261719,
            "n_vars": 100, // Inst. definition starts here
            "capacity": 23678,
            "profits": [
                // 50 integers, one for each p_i
            ],
            "weights": [
                // 50 integers, one for each w_i
            ]
        },
        (...) // More instances
        ,
        "n_solutions": 37,
    },
}}
```

Fig. 4. JSON file with the results from the instance generation experiment. This type of file is directly provided by DIGNEA.

MPI parallelism of the experiments and, as a result, it can be run in HPC systems to speed-up the process. For instance, DIGNEA has been deployed and executed correctly in various HPC systems such as Archer2[5] and TeideHPC.[6]

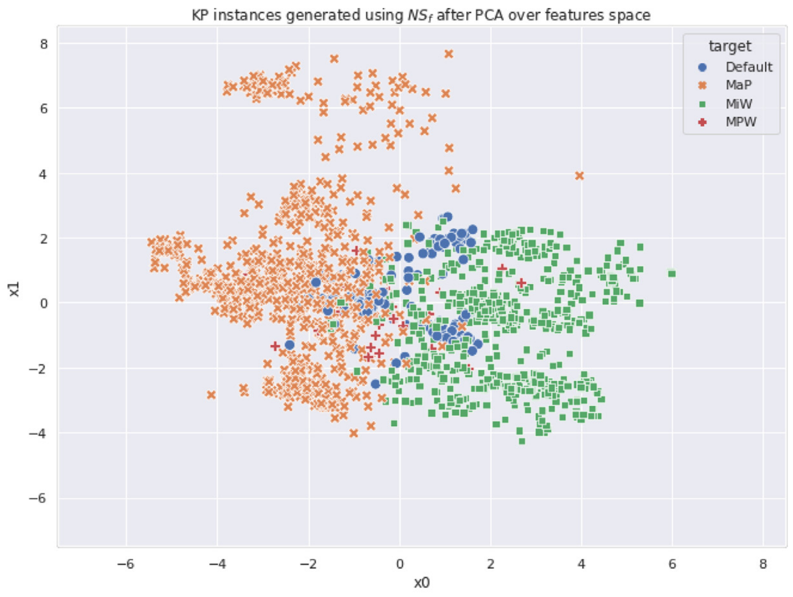
In a recent conference paper, DIGNEA was used to generate several sets of instances in the KP domain [13] for a set of different EA configurations. Results proved that the software is able to generate large sets of instances in a single run per target solver. Moreover, such instances demonstrated to be significantly biased to the performance of the target solver. In terms of space covering, the results outperformed other related proposals that rely on a pure evolutionary process with diversity management.

DIGNEA can be of great assistance to get more insight about problems, their instances, and how these instances share the feature and performance spaces with the aim of better design them. Furthermore, DIGNEA can be applied to pursue new research questions, for example, about instance space representation, instance characterisation via hand-designed features, automate designed features, algorithm design and selection or even parameter tuning evaluation for algorithms across domains. The software is now available as open source for the community.

5. Conclusions

We presented DIGNEA, a Diverse Instance Generator with Novelty Search and Evolutionary Algorithms. Our goal was to design a framework to generate instances in any optimisation domain. We achieved this by defining generic types and a modular architecture for DIGNEA. The current version of the software

[5] Archer2 website: https://www.archer2.ac.uk/.
[6] TeideHPC website: https://teidehpc.iter.es/.



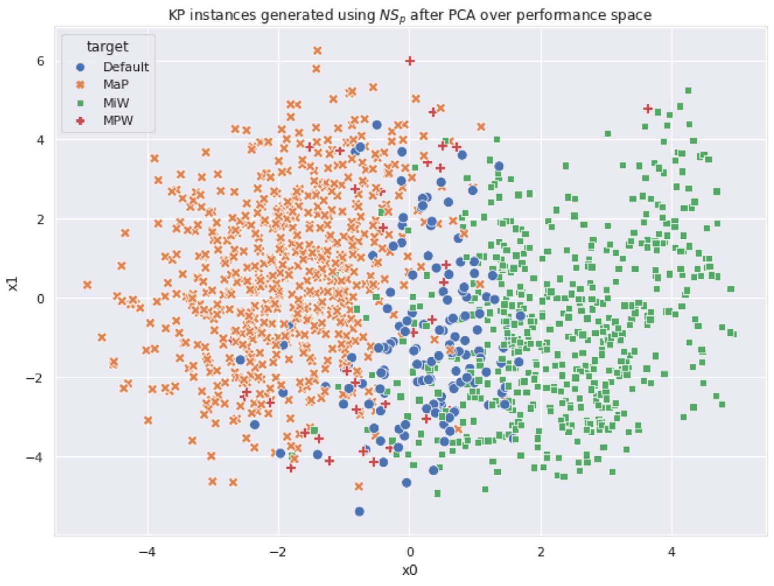


Fig. 5. Two-dimensional representation of KP instances generated through DIGNEA. Colours reflect the target algorithm for which an instance was produced. Instances on the top were generated using NSFeatures, while NSPerformance was used to generate instances shown at the bottom.

contains the required types for generating instances for the KP domain. Currently, we are working on including more domains, such as TSP and Bin-Packing.

The software has been used as the experimental framework for several conference papers and recent work. Results proved that it is able to generate better instances with respect to space coverage, novelty and fitness [13] in comparison to previous approaches which only considered random generation of maximisation of the performance gap among solvers [8,11].

Declaration of competing interest

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

Data availability

No data was used for the research described in the article.

Acknowledgements

The work of **Alejandro Marrero** was funded by the Canary Islands Government "Agencia Canaria de Investigación, Innovación *y* Sociedad de la Información - ACIISI" [contract number TESIS2020010005].

References

[1] Rice JR. The algorithm selection problem. Adv Comput 1976;15(C):65–118. http://dx.doi.org/10.1016/S0065-2458(08)60520-3.

[2] Kerschke P, Hoos HH, Neumann F, Trautmann H. Automated algorithm selection: Survey and perspectives. Evol Comput 2019;27(1):3–45. http://dx.doi.org/10.1162/evco_a_00242.

[3] Pisinger D. Where are the hard Knapsack problems? Comput Oper Res 2005;32(9):2271–84. http://dx.doi.org/10.1016/j.cor.2004.03.002.

[4] Smith-Miles K, Christiansen J, Muñoz MA. Revisiting where are the hard Knapsack problems? via instance space analysis. Comput Oper Res 2021;128:105184. http://dx.doi.org/10.1016/j.cor.2020.105184.

[5] Michalak K. Generating hard inventory routing problem instances using evolutionary algorithms. In: Proceedings of the Genetic and Evolutionary Computation Conference. 2021, p. 243–51. http://dx.doi.org/10.1145/3449639.3459401.

[6] Alissa M, Sim K, Hart E. Algorithm selection using deep learning without feature extraction. In: Proceedings of the Genetic and Evolutionary Computation Conference. 2019, p. 198–206. http://dx.doi.org/10.1145/3321707.3321845.

[7] 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. 2019, p. 58–71. http://dx.doi.org/10.1145/3299904.3340307.

[8] Plata-González LF, Amaya I, Ortiz-Bayliss JC, Conant-Pablos SE, Terashima-Marín H, Coello Coello CA. Evolutionary-based tailoring of synthetic instances for the Knapsack problem. Soft Comput 2019;23:12711–28. http://dx.doi.org/10.1007/s00500-019-03822-w.

[9] Julstrom B. Evolving heuristically difficult instances of combinatorial problems. In: Proceedings of the 11th annual conference on genetic and evolutionary computation. GECCO, ACM; 2009, p. 279–86. http://dx.doi.org/10.1145/1569901.1569941.

[10] Akgün Ö, Dang N, Miguel I, Salamon AZ, Stone C. Instance generation via generator instances. In: Schiex T, de Givry S, editors. Principles and practice of constraint. Cham: Springer International Publishing; 2019, p. 3–19. http://dx.doi.org/10.1007/978-3-030-30048-7_1.

[11] Ullrich M, Weise T, Awasthi A, Lässig J. A generic problem instance generator for discrete optimization problems. In: Proceedings of the Genetic and Evolutionary Computation Conference Companion. 2018, p. 1761–8. http://dx.doi.org/10.1145/3205651.3208284.

[12] Lehman J, Stanley KO. Abandoning objectives: Evolution through the search for novelty alone. Evol Comput 2011;19(2):189–222. http://dx.doi.org/10.1162/EVCO_a_00025.

[13] Marrero A, Segredo E, León C, Hart E. A novelty-search approach to filling an instance-space with diverse and discriminatory instances for the Knapsack problem. In: Parallel problem solving from nature – PPSN XVII. Cham: Springer International Publishing; 2022, p. 223–36. http://dx.doi.org/10.1007/978-3-031-14714-2_16.

[14] León C, Miranda G, Segura C. METCO: A parallel plugin-based framework for multi-objective optimization. Int J Artif Intell Tools 2009;18(04):569–88. http://dx.doi.org/10.1142/S0218213009000275.

[15] Marrero A, Segredo E, León 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. 2021, p. 1978–81. http://dx.doi.org/10.1145/3449726.3463160.