A Generic Problem Instance Generator for Discrete Optimization Problems

Markus Ullrich*
University of Applied Sciences
Zittau/Görlitz
Görlitz, Saxony, Germany
mullrich@hszg.de

Thomas Weise
Institute of Applied Optimization
Hefei University
Hefei, Anhui, China
tweise@hfuu.edu.cn

Abhishek Awasthi
University of Applied Sciences
Zittau/Görlitz
Görlitz, Saxony, Germany
aawasthi@hszg.de

Jörg Lässig
University of Applied Sciences
Zittau/Görlitz
Görlitz, Saxony, Germany
jlaessig@hszg.de

ABSTRACT

Measuring the performance of an optimization algorithm involves benchmark instances of related problems. In the area of discrete optimization, most well-known problems are covered by a large variety of problem instances already. However, while exploring the area of lesser-known optimization problems there is usually not a sufficient amount or variety of such benchmark instances available. The reasons for this lack of data vary from privacy or confidentiality issues to technical difficulties that prevent the collection of such data. This results in the inability to evaluate new optimization algorithms on these problems. Ideally, the necessary data for a variety of problem instances can be created randomly in advance to measure the performance of a set of algorithms. Random problem instance generators exist for specific problems already, however, generic tools that are applicable to multiple optimization problems are rare and usually restricted to a smaller subset of problems. We propose a generic problem instance generator for discrete optimization problems, which is easy to configure, and simple to expand to cover a broad variety of optimization problems. We show the capabilities of our tool by creating exemplary configurations for the TSP, Max-SAT and a real-world load allocation problem to generate random problem instances.

CCS CONCEPTS

• **Mathematics of computing → Optimization with randomized search heuristics**; **Evolutionary algorithms**; *Probabilistic algorithms*; Sequential Monte Carlo methods; • **Theory of computation → Random search heuristics**; *Theory of randomized search heuristics*;

*Corresponding Author

GECCO '18 Companion, July 15–19, 2018, Kyoto, Japan
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-5764-7/18/07…$15.00
https://doi.org/10.1145/3205651.3208284

ACM Reference Format:
Markus Ullrich, Thomas Weise, Abhishek Awasthi, and Jörg Lässig. 2018. A Generic Problem Instance Generator for Discrete Optimization Problems. In *GECCO '18 Companion: Genetic and Evolutionary Computation Conference Companion, July 15–19, 2018, Kyoto, Japan.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3205651.3208284

1 INTRODUCTION

The goal of this first instance of the Black-Box Discrete Optimization Benchmarking (BB-DOB) workshop is to define a suite of benchmark problems for black-box discrete optimization. A single-objective optimization problem is defined by a solution space $\mathbb{X}$ and an objective function $f : \mathbb{X} \mapsto \mathbb{R}$ mapping $\mathbb{X}$ to the real numbers $\mathbb{R}$ [18]. Usually, the goal is to find the value $x \in \mathbb{X}$ for which $f(x)$ takes on the smallest possible value. In the context of discrete (or combinatorial) optimization, $\mathbb{X}$ is finite [2, 12]. The sets of bit strings of a fixed length n or of the permutations of the first n natural numbers are common here.

An optimization algorithm is not applied to an optimization *problem* as a whole, but to a specific *instance*. For instance, we do not solve Traveling Salesmen Problems (TSP) [1, 4, 20] in general but the specific instance looking for the shortest tour through 14 cities in Burma [14]. After the problems have been chosen, several of their *instances* will be included into the BB-DOB benchmark set. The data of the selected instances should be directly available for any interested researcher. Interestingly, the history of optimization has shown that problem instances once considered to be hard can sometimes soon afterwards be solved to optimality.[1] In the ideal case, a researcher should thus not just have access to the data of the

[1]On the website http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/STSP.html listing the optimal solutions of the symmetric cases of TSPLib, we find: *"When I published TSPLIB more than 10 years ago, I expected that at least solving the large problem instances to proven optimality would pose a challenge for the years to come.*

selected instances, but also be able to generate more (in particular, larger and harder) problem instances. This leads to the dilemma that different researchers may use different instances and, hence, obtain incomparable results – or that each paper must be accompanied by additional files specifying the problem instances used.

We propose a simple way to solve this problem: A versatile problem instance generator which can produce instances of a very wide variety of discrete optimization problems and can do so *reproducibly*.

The rest of this paper is organized as follows. First, we discuss the related work on problem instance generation in Section 2. In Section 3, we then detail the requirements, specification, and utilization of our instance generator, highlighting the different data types and constraints that are supported. In Section 4, we provide three different examples of scenarios created by our generator. Finally, Section 5 concludes this paper and we discuss the planned future applications of our work.

2 RELATED WORK

A variety of well-known problems, such as the TSP [13], have been covered by problem instance generators already. However, these generators are usually problem-specific.

Drexl et al. [3], for instance, developed a problem instance generator for resource-constraint project scheduling problems (RCPSP) and several problem extensions. Their work is based on ProGen [8] an existing project generator which has been used previously for the RCPSP. From this example it becomes clear that it is not a simple task to create a general instance generator, even for this specific subset of problems. The dominant issue is the inability to predict future problems or problem extensions as described in the paper by Drexl et al. [3] which could not have been regarded during the development of ProGen. Our tool attempts to avoid this problem as it has been created with high flexibility in mind. Furthermore, it is easy to extend the generator which we will demonstrate for the creation of Max-SAT problem instances in Section 4.2.

However, to also highlight one of the limits of our generator, the work by Sakti et al. [15] is concerned with the generation of software test-data for object oriented code coverage. The challenges of this task cannot be met by our tool without spending a lot of additional work on creating an extension. Specifically, software test generators usually aim for a high code-coverage which cannot be achieved without meticulously analyzing the source code of a software. For these tasks, efficient tools like the generator presented by Sakti et al. [15] exist already.

Other applications for instance test generators include healthcare scheduling [9] in which only a few benchmark sets are known. The approach by Leeftink and Hans [9] relies on a case-mix classification scheme to create random instances based on existing data and measures their similarity to generate a highly diverse set of instances for benchmarking. This approach is different from our generator as it relies on existing data but, nonetheless, especially useful under these conditions. We believe that adding the ability to create a configuration from existing data to any generator will greatly enhance its capabilities.

However, due to enormous algorithmic progress all problems are now solved to optimality!!" (May 22, 2007)

To study the effect of non-regular problem instances, Macedo and Tchemisova [10] developed a generator for Semidefinite Programming (SDP) instances. This paper shows yet another useful application for problem instance generators. As non-regular problems can occur, it is valuable to be able to study the efficiency of popular solvers on these problems beforehand based on generated test data.

While related to our work, the presented instance generators in the above papers are tailor-made to fit a specific problem. Therefore, they are usually not applicable to different problems without major modifications. However, approaches that cover a variety of optimization problems exist as well. Hernando et al. [6] proposed a tunable instance generator. In their approach, the properties of the output instances are controlled by a set of parameters which allows the modification of their qualitative characteristics. Although this approach is applicable to a variety of problems it is specifically restricted to permutation-based combinatorial optimization problems.

Weise et al. [21] introduced a tunable benchmark model problem, the W-Model, for black-box benchmarking of black-box optimization algorithms. This problem allows for separately fine-tuning different problem difficulty characteristics, such as epistasis, neutrality/redundancy, and ruggedness/deceptiveness [19, 22]. These problems can also be turned into multi-objective tasks. Compared to our approach, the W-Model is limited to bit-string based search spaces. Its goal is not to provide instances for new real-world problems, but instead to investigate the impact of fitness landscape features on algorithm performance.

3 INSTANCE GENERATION APPROACH

In Section 1, we described how a typical problem is defined in our context. A problem instance in this regard is a collection of matrices in which values for a set of attributes and, in some cases, their relation to each other are represented, e.g., a distance matrix for a TSP containing the distance or travel time between cities to describe a problem instance. Therefore, a common and convenient way to store data from such a problem instance are CSV files and related formats like TSV.

Comprehensive resources including problem instances in such standardized formats exist already for well-known optimization problems. However, for lesser known or very specific problems, the lack of a sufficient amount of test data and thus problem instances can be an issue [9]. This holds especially for novel problems. The reasons for this lack of data vary from case to case and are either due to privacy or confidentiality issues or because the data for a problem instance has to be collected or generated first. While the first might prevent the publication of the data used in the experiments which is especially problematic in cases where obfuscating the data will destroy valuable information, the latter can be related to the scale of a dataset or a specific distribution of values that has not occurred yet in a real-world scenario but is likely to occur in the future. A usual solution in these cases is to create a problem specific instance generator [3, 9, 10, 13]. Some also offer the possibility to be fine-tuned which allows the creation of instances with varying difficulty [6, 19, 21, 22]. However, since those are usually directly tied to a specific problem they offer only limited possibilities for applications in other problems.

Considering that it is a time consuming process to create a generator for a single problem that presumably involves a lot of redundant tasks most of this effort could be avoided by creating a generic problem instance generator for not only existing but also future problems. Such a generator will take the definition or "blueprint" for one or multiple problem instances, e.g., problem attributes or variables and their relations to each other and create random or pseudo-random instances based on these definitions. For a generic instance generator as we described it these specifications should also be provided in a standardized manner to improve their readability and re-usability as well as the reproducibility of the resulting problem instances.

This Section describes the specific requirements for our generic problem instance generator based on the above problem formulation, each followed by their corresponding implementation details. Whereas Section 3.1 lists basic requirements, Section 3.2 describes how the generator can be configured and which options are available. We also highlight particular challenges regarding the verification process for the instance definitions in Section 3.3.

3.1 Requirements

The generator should use a standardized, human-readable format to specify the "blueprint" of the problem instances to be generated. We also wanted to use a lightweight format without much overhead. For these reasons, we chose JSON as it also offers enough flexibility to extend the generator with additional features if they become relevant in the future. The generator itself is written in Java and available on GitHub[2].

A second important requirement is that the generated instances should both be random and reproducible. This can be achieved by allowing the provision of the seeding value for the random number generator. Furthermore, the instances will also contain comments per default with related information such as the seeding used, the version of the generator, in case its value generation routine changes over time, and the Java version due to similar concerns about the random number generation process in Java.

Typically, an instance can be defined as a matrix. A vehicle routing problem may, for instance, specify several customers (rows) where each customer has a location (column 1), demand (column 2), and time window (column 3 and 4) for serving. Following this idea, the generator must allow specifying data types and constraints for columns as well as the number of rows to generate.

3.2 Generator Configuration

Table 1 shows a list of general configuration values that describe a problem instance. We are planning to add more options in the future to enable a more fine-grained tuning of the generator.

Since we plan to add new features for upcoming versions of the generator that older versions might ignore, the version for the generator that has to be used can be enforced with the *version* option. This setting is optional and serves currently just informational purposes since there is only one version to choose from which has to be deployed locally. However, for future development, we

Component	Example	Default	Optional
"version"	"v0.1"	"latest"	yes
"seed"	-5436791876534	random	yes
"rows"	100000	-	no
"problem"	VRP	-	yes
"no_duplicates"	true	false	yes
"print_parameters"	true	false	yes
"comments"	[· · ·]	-	yes
"parameters"	[· · ·]	-	yes
"attributes"	[· · ·]	-	no
"constraints"	[· · ·]	-	no

Table 1: Basic components of a problem instance "blueprint" for our generator.

Component	Example	Optional
"name"	"max_capacity"	no
er versions might ignore "type"	"double"	no
"value"	2000.00	no

Table 2: Basic components of a *parameter*

also would like to offer the generator as a web service with the ability to choose a different version with every request. Another option that helps to improve the reproducibility of results is *seed*, which can be used to provide the seeding for the random number generator. Other basic options that can be set are the number of *rows* which are generated in the resulting CSV file, whereas each row usually represents a datum of the problem instance, and the name of the *problem*, which will be printed as a comment in the results if provided. Finally, there are optional flags that can be set. The option *no_duplicates* set to true enforces that none of the rows which are generated for a problem instance are exactly the same. The flag *print_parameters* enables that optional problem parameters, which can be defined separately, are printed as a comment in the results as well. The last part of the configuration file are lists of *comments*, which will be included in the result, problem specific *parameters*, which only have a cosmetic function for now, if the *print_parameters* flag is set to true, *attributes* and *constraints*.

Whereas *comments* are quite simply a list of *comment* values, every other list consists of complex elements which we are going to describe in more detail now. Table 2 lists the currently supported options for a *parameter*. The current version of the generator supports simple values only, which means all of the three options have to be present. The *name* represents the identifier of this parameter, the *type* option can be set to one of the following: *{integer, double, boolean, nominal}* and the *value* must match the specified type, e.g., 0.12 for *double* and "string" for *nominal* types.

For *attributes* more complex options exist as shown in Table 3. Similar to a parameter, an identifier can be provided with the *name* option. *Type* and *value* can be set as well, however, providing a single value for an attribute is optional and not recommended. For most types additional settings exist to define the range of values instead. If a *value* is not provided at least one of the following options has to be present. One possibility is to provide *min* and *max* values for numeric types including an optional *default* value.

[2]https://github.com/mullrichHSZG/BBDOB_problem-generator

Component	Example	Default	Optional
"name"	"x"	-	no
"type"	"double"	-	no
"value"	726.13	0.0	yes
"min"	0.0	0.0	yes
"max"	9999.99	1.0	yes
"default"	500.0	-	yes
"mean"	500.0	0.0	yes
"standard_deviation"	200.0	1.0	yes
"true"	0.7	0.5	yes
"start"	1000	0	yes
"increment"	1000	1	yes
"minIncrement"	10	1	yes
"maxIncrement"	100	1	yes
"stop"	10000	-	yes
"expression"	"5 * y"	-	yes
"use_all_values"	true	false	yes
"output"	true	true	yes
"seed"	9354673	global_seed	yes

Table 3: Basic components of an *attribute*

Furthermore, the *type* can be set to *Gaussian*, which is another numeric type for attributes to generate normal distributed values using the Polar method by Marsaglia and Bray [11]. The *mean* and *standard_deviation* can be specified additionally. In case *boolean* has been chosen as a type, the percentage in which the value evaluates to true can be modified with the *true* option. If an id value needs to be generated, the *id* type can be used which generates integer values starting with 0 and using an increment of 1 for every additional value. Optional settings for this type are a different *start* value and the *increment* for every row. For the latter, a *minIncrement* and *maxIncrement* value can be specified alternatively which causes the generator to randomly pick the increment from these boundaries for every row. Furthermore, only the *minIncrement* option can be provided along with a *stop* value. This will cause the generator to produce a random sequence between the *start* and *stop* value using as many steps as the number of rows with a value of at least *minIncrement*. Lastly, instead of *value*, *min* or *max*, an *expression* can be specified for any numeric type which will be evaluated during the generation process. The example expression in Table 3 will cause the value of x to always be 5 times the value of y in every row of the output file. For nominal types no specific options exist currently.

Besides type specific options, more general options are available as well. For instance, it is possible to provide a local random seed for every attribute. If a local seed is not provided, the global seed will be used instead. If a local seed is specified for every attribute, the global seed will not be used at all since all random numbers are generated locally. If the flag *use_all_values* is set to true the generator will ensure that every possible value for this attribute is generated at least once, e.g., for an integer attribute with $min = 1$ and $max = 5$, each value in {1,2,3,4,5} is generated at least once for this attribute. The method we utilize to guarantee this behavior verifies that the number of possible values does not exceed the number of rows first. Then we insert all possible values in a list with a length equal to the number of rows. After that, all remaining empty spots are filled with pre-generated random numbers and the list is shuffled using the *Collections.shuffle()* method provided by Java. If during the generation process one or several constraints cannot be fulfilled using the current next value from such a pre-generated list the remaining list will be re-shuffled. It is recommended that no or only a few other constraints, in which this attribute is restricted by a different attribute, are present in the configuration. Finally, the *output* flag can be modified as well for every attribute. If set to true, which is the default, the attribute is part of the output CSV file, otherwise it will not be printed. This might be particularly useful for attributes that use an expression to support the definition of complex constraints. We explain this procedure briefly in an upcoming paragraph.

Component	Example	Optional
"name"	"x!=y"	no
"left"	"x"	no
"relation"	"!="	no
"right"	"y"	no

Table 4: Basic components of a *constraint*

Lastly, *constraints* represents an array of restrictive properties of the resulting problem instance which are checked on a row by row basis during the generation process. Every constraint consists of four components which are listed in Table 4. As with parameters and attributes, constraints have to have a unique identifier provided with the *name* option. Further, a constraint consists of two expressions, *left* and *right*, and their *relation* to each other. An expression can be - similar to an expression that can be provided for a numeric attribute - any kind of arithmetic expression, e.g., a constant, an attribute or an addition, subtraction, etc. Statistical expressions, e.g., min, max, mean or the sum of all values of an attribute that have been created so far can be used as well. Table 5 shows a list of all possible expressions and provides an example configuration each, which consists of a type and its value. Regarding their relation, currently, only constraints in the form of a linear inequality with integer or rational domains are supported. Equalities are allowed as well, but those should be used sparingly since they are highly restrictive which can lead to an empty domain for an attribute and thus no valid problem instance can be created. Generally, only simple expressions, as the ones described above, can be used. However, complex expressions, e.g., $\frac{x_2 \cdot 2}{y_2}$, can be created using additional attributes. To create the previous example as an expression for a constraint, one would first have to create an attribute with the expression "x2 * 2" as its value and give it a name. We call this attribute "z" for now. The *output* value can be set to "false" so this attribute will not be present in the generated CSV file. The expression for the constraint can now be provided with "z / y2". In the future, we are also planning to automate this process which means complex expressions can be provided without the manual creation of a support variable. The problem generator itself will take care of creating the necessary attribute.

Expression	Type	Example Value
Constant	"integer"	300
Variable	"attribute"	"x1"
Sum	"expression"	"sum(x2)"
Min	"expression"	"min(y1)"
Max	"expression"	"max(y2)"
Mean	"expression"	"avg(i)"
Absolute	"expression"	"abs(j)"
Addition	"expression"	"j + 3"
Subtraction	"expression"	"100 - x1"
Multiplication	"expression"	"y2 * x2"
Division	"expression"	"x1 / 2"

Table 5: List of expressions that are currently supported by the problem generator

3.3 Constraint Validation

Before a problem instance will be created, the generator attempts to validate the constraints that have been provided by checking for circular dependencies and general problems with the configuration.

First, we check for attribute boundaries that make it impossible to fulfill certain constraints, e.g., $max(x) = 10$ and $min(y) = 20$ with the constraint $x \geq y$. To check if the *no_duplicates* constraint can be fulfilled, the number of possible combinations of values will be calculated. This number must be higher or equal to the number of generated rows in the output file. The current implementation only takes the boundaries that are provided with the definition of the attributes into account, e.g., min and max values, and excludes attributes for which *output* is set to false. Further, the attributes are sorted based on their dependencies on each other by using the method developed by Kahn for topological sorting of nodes [7]. That means, in order to sort the attributes they are placed as nodes in a directed graph whereas the constraints define the edges for that graph going from *right* to *left*. Since an expression that has been directly provided for an attribute might represent an additional dependency it can also be interpreted as an edge in the graph. Expressions with a constant value are not considered during this process. Sorting the attributes using this method is also necessary for the generation process. To ensure that all dependencies for every attribute can be met we simply generate the values in the topological order of the directed graph, starting with all attributes that have no incoming edges.

Once the generation has been started, the generator will check the validity of every value that is created during the process and generate new values in case a constraint has not been fulfilled.

4 EXAMPLE SCENARIOS

In this Section we provide three example configurations for the problem instance generator. First, we are focusing on well-known problems, e.g., TSP and the general Max-SAT problem in Sections 4.1 and 4.2 respectively. After that, we introduce a novel problem which is based on the resource constraint project scheduling problem with time window. Section 4.3 describes the real-world load allocation problem, explains the necessity to generate data and how an example configuration for our generator could be used to create various problem instances.

Listing 1: TSP Configuration for the Generator

```
{
  "version": "v0.1",
  "seed": -198465108435,
  "rows": 1000,
  "problem": "TSP",
  "no_duplicates": true,
  "attributes": [
    {  "name": "id",
       "type": "id",
       "start": 1,
       "increment": 1 },
    {  "name": "x",
       "type": "double",
       "min": 0.0,
       "max": 10000.0 },
    {  "name": "y", ... }
  ],
  "constraints": []
}
```

Listing 2: TSP Example Result

```
# comments omitted for brevity
id;x;y
1;9815.99;6830.20
2;3021.45;4173.25
3;4185.60;8340.02
...
998;8337.32;7522.33
999;6650.97;2555.37
1000;7526.32;3451.16
```

4.1 TSP Example Scenario

We choose TSP as our first example since its input format, although several variations exist, is already very-well defined. Furthermore, it has a very simple and human readable structure which makes it a perfect candidate to show the basic capabilities of the generator.

The example configuration for the TSP problem instance format we decided to use is shown in Listing 1. The attributes we define are a unique identifier for every location that can be visited and x and y coordinates on a two-dimensional euclidean plane. Constraints are not necessary in this case. We further provide a seed to ensure that the same instance will be generated every time.

The output in Listing 2 is roughly equivalent to the uniformly distributed instances from the DIMACS TSP challenge [4]. While instances like these, as well as random distance matrices can be produced by our current generator, instances with clustered cities are not yet possible: They could be modeled using more complex probability distributions for attribute values, such as multiple two-dimensional normal distributions, which we will include in the near future.

4.2 Max-SAT Example Scenario

To show the flexibility and extensibility of the generator, we choose Max-SAT as our second example. Specifically, we want to generate problem instances according to the CNF specification in the DIMACS format[3]. The configuration in Listing 3 uses additional

[3]http://maxsat.ia.udl.cat/requirements/ - last visit: 01.04.2018

Listing 3: Max-SAT Example Configuration based on the CNF Standard

```
{ ...
  "separator": " ",
  "comment_prefix": "c",
  "alternative_header": "p cnf 4 100",
  "attributes": [
    { "name": "i",
      "type": "integer",
      "min": -4,
      "max": 4 },
    { "name": "j", ... },
    { "name": "k", ...,
      "output_probability": 0.3 }
    { "name": "zero",
      "type": "integer",
      "value": 0}
  ],
  "constraints": [
    { "name": "i!=j",
      "left": {
        "type": "attribute",
        "value": "i" },
      "relation": "!=",
      "right": {
        "type": "attribute",
        "value": "j" } },
    { "name": "k!=j", ... },
    { "name": "k!=i", ... },
    { "name": "no_i_zero",
      "left": {
        "type": "attribute",
        "value": "i" },
      "relation": "!=",
      "right": {
        "type": "integer",
        "value": 0 } },
    { "name": "no_j_zero", ... },
    { "name": "no_k_zero", ... } }
  ]
}
```

settings for the generator that we have not described yet, however, most of them are only used for cosmetic reasons. For instance, we provide an *alternative_header*, a different *comment_prefix* and change the *separator* value from the default ";" to adhere with the required problem instance format. In our example we want to create a problem instance with 100 clauses and 4 variables. The latter are represented in the clauses using the i, j and k attributes with the boundaries −4 and 4. To create clauses of varying length we added the ability to change the *output_probability* of an attribute. In this case, the column with the k attribute will only be printed in 30% of the rows. The *zero* attribute is required by the standard to be present at the end of every clause. The constraints ensure that the absolute values of the attributes which are present in every clause are distinct and non zero. By using the absolute value, we also prevent clauses that are always true and thus irrelevant, e.g., $(x_1 \vee \neg x_1)$.

Listing 4 shows the output of the generator for the Max-SAT configuration we provided earlier. Although the problem instance definition is rather long compared to the TSP, we believe that the creation process is very straightforward and the file is human readable which was one of our main concerns for creating this generator.

Listing 4: Max-SAT Example Result

```
c comments omitted for brevity
p cnf 4 100
-1 3 0
4 -2 0
-2 1 -3 0
...
4 -1 0
3 -2 0
```

4.3 Load Allocation Example Scenarios

For our third example, we choose a novel problem with a real-world application case. This specific problem is part of a current research project which is concerned with the development and evaluation of a management platform for the distribution grid and has, to the best of our knowledge, not been published yet.

For the discrete load allocation problem, a certain amount of power is generated by one or multiple power-supplier(s) for a given period of time, to meet the demands of the consumers. The inputs for the problem are the power-schedule as a step function with respect to time, and the sum consumer requirements-schedule for the demand with a flexible time window for the power delivery denoted by the earliest and latest possible power feeding time for every individual consumer. In our application scenario, a single supplier exists for multiple consumers. If the supplier is not able to deliver all of the generated power to the customers, a penalty per time unit is incurred by the supplier. Likewise, if the power consumption of a consumer exceeds the total power availability according to the schedule at any time t, a penalty is incurred by the supplier as well as he has to buy additional resources at the intra-day market for a higher price. Figure 1 shows one possible solution for this problem with the supply for a single supplier and the demand as a sum of the power consumption of all customers. The optimization problem is a modification of the project scheduling and standard resource allocation scheduling problem [5, 16].

Let

n	=	the total number of customers to whom the power is to be delivered,
P_t	=	the power available at time t, $P_t \geq 0 \ \forall \ t$,
τ_i	=	total time for which customer i needs power,
d_i	=	power demand by customer i for a period of time τ_i, $\tau_i \geq 0 \ \forall \ i$,
θ_i^t	=	power consumption of a customer i at any time t,
E_i	=	earliest possible power feeding time to customer i,
L_i	=	latest possible power feeding time to customer i, $L_i - E_i \geq \tau_i$,
α^t	=	penalty incurred by the supplier at any time t, for not meeting the total requirement of the consumers,
β_i	=	penalty associated with any customer i, for exceeding the power consumption,
s_i	=	start time for power feeding to customer i.

With the above formulation, it is clear that $\sum_{t=s_i}^{s_i+\tau_i} \theta_i^t = d_i$. The objective of the problem is to find the feasible starting times of the

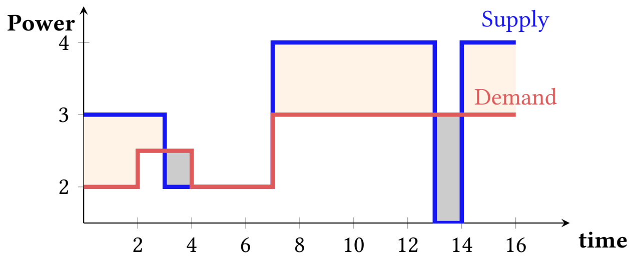

Figure 1: The figure showing the demand and supply schedules. The light region between the schedules is the total amount of excess supply from the supplier. The dark region is the excess demand from the customers.

power consumption by the customers, to reduce the sum of the total weighted penalties incurred by the customer and the supplier. The constraint of the earliest and the latest power feeding times to any customer, must be respected. The objective function for this discrete optimization problem can be formally expressed as

$$\min \sum_t \alpha^t \cdot \max \left\{ 0, P_t - \sum_i \theta_i^t \right\} + \sum_i \sum_t \beta_i \cdot \max \left\{ 0, \theta_i^t - P_t \right\} \ . \tag{1}$$

According to the problem formulation, at least two different files need to be created for a valid problem instance. One file has to describe the power demand of every customer. In our application scenario, only a single supplier exists, so only one other file is necessary that describes the schedule of the power supplier. From the problem description we derived the configurations for the customers and the supplier which are shown in Listings 5 and 6 respectively.

We believe that, for the most part, theses configuration files are self-explanatory. Therefore, there are only a few details we would like to point out. The constraint in the customer configuration file is the representation of the mathematical notation $L_i - E_i \geq \tau_i$ and ensures that the required power can be delivered during the given time window for every customer. The schedule is provided with indexes that each represent a 15 minute time interval during one day followed by the change in power supply. The increment from one index to another can be random. The supply is assumed to remain constant in between these intervals. Therefore, we used the *id* data type with a random step interval and a *stop* value at 95 to ensure that the generated indexes match the problem description. We also provided a global and sometimes local seeds, which we omitted for brevity, in each of the configurations to ensure result reproducibility. Listings 7 and 8 show the generated files for the customers and the supplier respectively.

5 CONCLUSION AND FUTURE WORK

We developed a generic problem instance generator for discrete optimization problems. This generator can be used to create problem instances for a large variety of discrete optimization problems in an efficient manner. We showed the flexibility of our tool by generating instances of well-known optimization problems, i.e., TSP and Max-SAT, and for a real-world load allocation problem based on the resource constraint project scheduling with time window problem. We demonstrated how a formal problem description can be translated into a configuration file for our generator.

Listing 5: Load Allocation Configuration for the Customers

```
{  ...
  "rows": 10,
  "problem": "LoadAllocation_Customer",
  "no_duplicates": true,
  "attributes": [
  { "name": "n",
    "type": "id",
    "start": 1 },
  { "name": "E_i",
    "type": "integer",
    "min": 0,
    "max": 70 },
  { "name": "L_i",
    "type": "integer",
    "min": 20,
    "max": 95 },
  { "name": "tau_i",
    "type": "integer",
    "min": 2,
    "max": 20 },
  { "name": "d_i",
    "type": "integer",
    "min": 1,
    "max": 10 },
  { "name": "beta_i",
    "type": "integer",
    "min": 1,
    "max": 10 }
  ],
  "constraints": [
  { "name": "L_i >= E_i + tau_i",
    "left": {
      "type": "attribute",
      "value": "L_i" },
    "relation": ">=",
    "right": {
      "type": "expression",
      "value": "E_i + tau_i" } }
  ]
}
```

Listing 6: Load Allocation Configuration for the Supplier

```
{  ...
  "rows": 10,
  "problem": "LoadAllocation_Supplier",
  "no_duplicates": true,
  "attributes": [
  { "name": "t",
    "type": "id",
    "start": 0,
    "minIncrement": 2,
    "stop": 95 },
  { "name": "P_t",
    "type": "integer",
    "min": 0,
    "max": 10 },
  { "name": "alpha_t",
    "type": "integer",
    "min": 1,
    "max": 10 }
  ],
  "constraints": []
}
```

Listing 7: Load Allocation Customers Example Result

```
# comments omitted for brevity
n;E_i;L_i;tau_i;d_i;beta_i
1;70;81;5;7;3
2;4;40;10;6;4
...
9;41;91;13;5;6
10;62;69;6;10;9
```

Listing 8: Load Allocation Supplier Example Result

```
# comments omitted for brevity
t;P_t;alpha_t
0;0;2
9;7;8
...
72;7;2
95;1;5
```

While we were aiming to build the generator as generic as possible, it still creates random instances. Therefore, the quality characteristics of these instances cannot be assessed prior to benchmarking. Although the creation of complex constraints can reduce the impact of this limitation to some extent, the randomly generated instances will not reach the quality of carefully selected test instances.

Nonetheless, the focus of the first instance of the BB-DOB workshop is to define a set of problems for black-box discrete optimization benchmarking. Of course, for each problem, several instances are required. We think that it would be suitable for creating the instances of at least some of the problems that will be chosen into the final BB-DOB benchmark set. This way, one could first obtain an initial set of "default" benchmark instances. As stated in the introduction Section 1, however, we can hope that these instances can eventually be solved efficiently in the future, even if including hard problems for today's approaches. Having a standardized, well-defined, documented, open source software process to create new instances will allow us to create new, harder, or larger instances. Due to the availability of our generator, any researcher can do so and make these instances accessible to others by including the configuration script in their publication, rather than potentially hundreds of files.

In the future, we would like to further expand the capabilities of our tool. As we already mentioned, the support for complex expressions and constraints will be added to increase the readability of the configuration. Furthermore, different types of constraints should be supported as well, e.g., satisfaction problems as well as regular expressions for nominal values. Another feature that could improve the usability would be the automatic or bulk generation of attributes to simplify the generation of matrices.

To improve the re-usability of a "blueprint" we will also add the ability to pass certain parameters on the command line which will take precedence over the values in the configuration file. Then the same file can then be reused with slightly different parameters, e.g., the seeding for the random number generator or the number of rows. This will also be useful for the bulk generation of problem instances. Lastly, although a lot of manual testing has been conducted, we are currently implementing unit tests for the different aspects of the generation process to further increase the confidence in the correctness of the code and generated instances.

ACKNOWLEDGEMENTS

Dr. Weise acknowledges support by the National Natural Science Foundation of China under Grants 61673359, 61150110488, and 71520107002.

REFERENCES

[1] David Lee Applegate, Robert E. Bixby, Vašek Chvátal, and William John Cook. 2007. *The Traveling Salesman Problem: A Computational Study.* Princeton University Press, Princeton, NJ, USA.

[2] William J. Cook, William H. Cunningham, William R. Pulleyblank, and Alexander Schrijver. 1997. *Combinatorial Optimization.* Wiley Interscience, Chichester, West Sussex, UK.

[3] Andreas Drexl, Rüdiger Nissen, James H. Patterson, and Frank Salewski. 2000. ProGen/πx - An instance generator for resource-constrained project scheduling problems with partially renewable resources and further extensions. *European Journal of Operational Research* 125, 1 (2000), 59 – 72. https://doi.org/10.1016/S0377-2217(99)00205-2

[4] Gregory Z. Gutin and Abraham P. Punnen (Eds.). 2002. *The Traveling Salesman Problem and its Variations.* Combinatorial Optimization, Vol. 12. Kluwer Academic Publishers, Berlin, Heidelberg. https://doi.org/10.1007/b101971

[5] S. Heinz and J.C. Beck. 2011. Solving resource allocation/scheduling problems with constraint integer programming. In *Proceedings of the Workshop on Constraint Satisfaction Techniques for Planning and Scheduling Problems (COPLAS).* 23–30.

[6] L. Hernando, A. Mendiburu, and J. A. Lozano. 2016. A Tunable Generator of Instances of Permutation-Based Combinatorial Optimization Problems. *IEEE Transactions on Evolutionary Computation* 20, 2 (April 2016), 165–179. https://doi.org/10.1109/TEVC.2015.2433680

[7] A. B. Kahn. 1962. Topological Sorting of Large Networks. *Commun. ACM* 5, 11 (Nov. 1962), 558–562. https://doi.org/10.1145/368996.369025

[8] Rainer Kolisch, Arno Sprecher, and Andreas Drexl. 1995. Characterization and Generation of a General Class of Resource-Constrained Project Scheduling Problems. *Management Science* 41, 10 (1995), 1693–1703. https://doi.org/10.1287/mnsc.41.10.1693 arXiv:https://doi.org/10.1287/mnsc.41.10.1693

[9] Gréanne Leeftink and Erwin W. Hans. 2018. Case mix classification and a benchmark set for surgery scheduling. *Journal of Scheduling* 21, 1 (01 Feb 2018), 17–33. https://doi.org/10.1007/s10951-017-0539-8

[10] Eloísa Macedo and Tatiana Tchemisova. 2017. A Generator of Nonregular Semidefinite Programming Problems. In *Springer Proceedings in Mathematics and Statistics*, Springer (Ed.). http://hdl.handle.net/10773/18252

[11] G. Marsaglia and T. A. Bray. 1964. A Convenient Method for Generating Normal Variables. *SIAM Rev.* 6, 3 (1964), 260–264. https://doi.org/10.2307/2027592

[12] Panos M. Pardalos and Ding-Zhu Du (Eds.). 1990. *Handbook of Combinatorial Optimization.* Vol. 1–3. Kluwer Academic Publishers, Norwell, MA, USA.

[13] Ronald L. Rardin, Craig A. Tovey, and Martha G. Pilcher. 2011. *Analysis of a Random Cut Test Instance Generator for the TSP.* WORLD SCIENTIFIC, 387–405. https://doi.org/10.1142/9789814354363_0017

[14] Gerhard Reinelt. 1991. TSPLIB – A Traveling Salesman Problem Library. *ORSA Journal on Computing* 3, 4 (Fall 1991), 376–384. https://doi.org/10.1287/ijoc.3.4.376 Instances: http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/tsp/, accessed 2017-12-28.

[15] A. Sakti, G. Pesant, and Y. G. Guéhéneuc. 2015. Instance Generator and Problem Representation to Improve Object Oriented Code Coverage. *IEEE Transactions on Software Engineering* 41, 3 (March 2015), 294–313. https://doi.org/10.1109/TSE.2014.2363479

[16] T.B. Smith and M.P. John. 2004. An effective algorithm for project scheduling with arbitrary temporal constraints.. In *Proceedings of Association for the Advancement of Artificial Intelligence (AAAI).* 544–549.

[17] Markus Ullrich, Thomas Weise, Abhishek Awasthi, and Jörg Lässig. [n. d.]. A Generic Problem Instance Generator for Discrete Optimization Problems. In *Black Box Discrete Optimization Benchmarking (BB-DOB) Workshop in Companion Material Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2018), July 15–19, 2018, Kyoto, Japan.* ACM, 1761–1768. https://doi.org/10.1145/3205651.3208284

[18] Thomas Weise. 2009. *Global Optimization Algorithms – Theory and Application.* it-weise.de (self-published), Germany. http://www.it-weise.de/projects/book.pdf

[19] Thomas Weise, Raymond Chiong, and Ke Tang. 2012. Evolutionary Optimization: Pitfalls and Booby Traps. *Journal of Computer Science and Technology (JCST)* 27 (2012), 907–936. Issue 5. https://doi.org/10.1007/s11390-012-1274-4

[20] Thomas Weise, Raymond Chiong, Ke Tang, Jörg Lässig, Shigeyoshi Tsutsui, Wenxiang Chen, Zbigniew Michalewicz, and Xin Yao. 2014. Benchmarking Optimization Algorithms: An Open Source Framework for the Traveling Salesman Problem. *IEEE Computational Intelligence Magazine (CIM)* 9, 3 (Aug. 2014), 40–52. https://doi.org/10.1109/MCI.2014.2326101

[21] Thomas Weise, Stefan Niemczyk, Hendrik Skubch, Roland Reichle, and Kurt Geihs. 2008. A Tunable Model for Multi-Objective, Epistatic, Rugged, and Neutral Fitness Landscapes. In *Genetic and Evolutionary Computation Conference*. ACM Press, Atlanta, GA, USA, 795–802.

[22] Thomas Weise, Michael Zapf, Raymond Chiong, and Antonio Jesús Nebro Urbaneja. 2009. Why is optimization difficult? In *Nature-Inspired Algorithms for Optimisation*, Raymond Chiong (Ed.). Studies in Computational Intelligence, Vol. 193. Springer-Verlag, Berlin/Heidelberg, Chapter 1, 1–50. https://doi.org/10.1007/978-3-642-00267-0_1

```
@inproceedings{UWAL2018AGPIGFDOP,
  author = {Markus Ullrich and Thomas Weise and
            Abhishek Awasthi and J{\"o}rg L{\"a}ssig},
  title = {A Generic Problem Instance Generator for Discrete
           Optimization Problems},
  booktitle = {Black Box Discrete Optimization Benchmarking
               (BB-DOB) Workshop in Companion Material
              Proceedings of the Genetic and Evolutionary
              Computation Conference (GECCO 2018), } #
              jul # {~15--19, 2018, Kyoto, Japan},
  pages = {1761--1768},
  publisher = {ACM},
  ISBN = {978-1-4503-5764-7},
  doi = {10.1145/3205651.3208284},
}
```