# A Framework for Generating Informative Benchmark Instances

**Nguyen Dang**
School of Computer Science, University of St Andrews, United Kingdom

**Özgür Akgün**
School of Computer Science, University of St Andrews, United Kingdom

**Joan Espasa**
School of Computer Science, University of St Andrews, United Kingdom

**Ian Miguel**
School of Computer Science, University of St Andrews, United Kingdom

**Peter Nightingale**
Department of Computer Science, University of York, United Kingdom

## Abstract

Benchmarking is an important tool for assessing the relative performance of alternative solving approaches. However, the utility of benchmarking is limited by the quantity and quality of the available problem instances. Modern constraint programming languages typically allow the specification of a class-level model that is parameterised over instance data. This separation presents an opportunity for automated approaches to generate instance data that define instances that are *graded* (solvable at a certain difficulty level for a solver) or can *discriminate* between two solving approaches. In this paper, we introduce a framework that combines these two properties to generate a large number of benchmark instances, purposely generated for effective and informative benchmarking. We use five problems that were used in the MiniZinc competition to demonstrate the usage of our framework. In addition to producing a ranking among solvers, our framework gives a broader understanding of the behaviour of each solver for the whole instance space; for example by finding subsets of instances where the solver performance significantly varies from its average performance.

**2012 ACM Subject Classification** Theory of computation → Constraint and logic programming

**Keywords and phrases** Instance generation, Benchmarking, Constraint Programming

**Digital Object Identifier** 10.4230/LIPIcs.CP.2022.26

**Supplementary Material** *Code*: https://github.com/stacs-cp/AutoIG

**Funding** *Nguyen Dang*: is a Leverhulme Early Career Fellow
*Ian Miguel*: supported by EPSRC EP/V027182/1

**Acknowledgements** This work uses the Cirrus UK National Tier-2 HPC Service at EPCC (http://www.cirrus.ac.uk) funded by the University of Edinburgh and EPSRC (EP/P020267/1).

## 1 Introduction

A practitioner faced with solving a new problem has a difficult choice among many solving algorithms, whose performance on the new problem is unknown and is likely to be variable. One approach is to draw instances from the problem to *benchmark* the various solvers under consideration, i.e. an empirical study of relative performance. This approach is favoured for computationally challenging tasks since the performance behaviour of a non-trivial algorithm is difficult to predict and is unlikely to be susceptible to a purely theoretical analysis [10]. As Beiranvand et al. [11] argue, care must be taken to select an instance set with a variety of difficulty for benchmarking in order to obtain the best insight into solver performance.

© Nguyen Dang, Özgür Akgün, Joan Espasa, Ian Miguel, Peter Nightingale;
licensed under Creative Commons License CC-BY 4.0

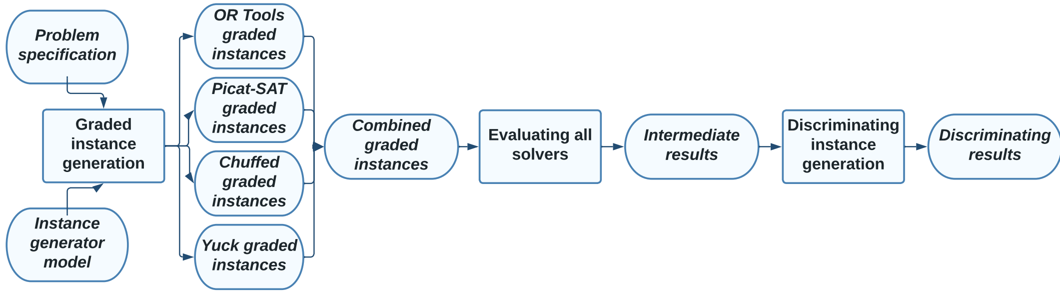


**Figure 1** Flowchart of the whole AutoIG application process

Constraint programming (CP) approaches particularly benefit from empirical analysis, since modern tool chains like MiniZinc [34] and savilerow [35] support targeting multiple solvers from a solver-independent constraint model. These may be entirely different paradigms, such as SAT [13], SMT [9] or indeed CP, and so can vary in performance significantly.

The need for empirical benchmarking is further supported by competitions run by several research communities, like the MiniZinc challenge [44] in the CP community, the SAT competition [18] and the AI planning competition [47]. Solver developers enter a competition by providing a default configuration of their solver. Each solver supports a common interface for specifying their input and output. The competition is then run on a set of problem instances and the solvers are ranked with respect to their comparative performance.

In the main solver competition for CP, the MiniZinc challenge, each solver is given two inputs: a solver-independent problem-level model and instance data written in a separate data file. Then MiniZinc is used to instantiate and translate the solver-independent model into input suitable for each solver. The main result of the challenge is a ranking of solvers. More detailed results pertaining to the ranking of solvers per problem class are also published.

The selection of problem instances to be used in a competition is extremely important to avoid conclusions that are unintentionally biased towards the chosen instances. Competitions somewhat mitigate this problem by inviting solver authors to submit benchmark instances. This is a promising sociotechnical attempt at alleviating the problem of bias, but it is laborious and does not provide a comprehensive solution.

Benchmarking is not only useful for finding an overall ranking among options, but also for finding subsets of instances where the performance of a solver is significantly different from the performance of the same solver overall. For example, solver $A$ might perform better for most instances of a problem class in comparison with $B$, yet perform very poorly for a particular subset of the instances. Information like this can be extremely valuable to solver developers. A traditional competition that works by running all solvers on a fixed set of instances can occasionally detect such cases even though it does not actively look for them.

For an informative benchmark we need a sufficient *quantity* of high *quality* instances and the ability to dynamically explore subsets of the instance space to detect performance discrimination. In this work we present AutoIG, a constraint-based instance generation framework, that supports automatically generating *graded* instances (i.e., solvable at a certain difficulty level for a given solver), and finding discriminating instances (i.e. easy for one solver and difficult for another solver). In combination, these two methods can be used to generate a large number of high-quality instances. Furthermore, they can be used to find interesting subsets of the instance space as opposed to leaving their discovery to chance.

Figure 1 gives a flowchart for an end-to-end application of AutoIG, whose instance generation process is explained in Section 3. Without loss of generality, the flowchart lists the

four solvers used for the evaluation of AutoIG in this paper. Section 4 explains the choice of these solvers and the five problem classes we use. Both stages of AutoIG can be applied to other solvers and solver configurations. The AutoIG process has two main inputs: a problem specification (in the form of a MiniZinc model in this paper) and a problem specific instance generator. The instance generator is parameterised to allow AutoIG to generate a variety of instances. There are two main places where we can extract results from AutoIG, evaluating all solvers on the combined set of graded instances (marked intermediate results in the flowchart, see Section 6) and evaluating the results of discriminating instances (marked discriminating results in the flowchart, see Section 7). AutoIG source code and all data and models used in this paper are available at `https://github.com/stacs-cp/AutoIG`.

The main contributions of this paper include:

1. A novel constraint-based framework for generating informative benchmark instances which combines two approaches (*graded* and *discriminating* instance generation) that were previously used in isolation [4, 3].
2. Support for MiniZinc and hand-written instance generators. The new system accepts a user-defined generator as a constraint model, thus allowing problem-specific knowledge to be injected into the instance generation process.
3. Support for the evaluation of local search solvers in addition to systematic solvers. The instance evaluation also considers both solution quality and running time.
4. An extensive evaluation on five problems from the MiniZinc challenge, showing that we can gain new interesting insights that complement the competition's results.

## 2 Related Work

A series of papers uses evolutionary algorithms and applies *instance space analysis* methods to problems in machine learning (classification [32], regression [33], clustering [17]) and in combinatorial optimisation (personnel scheduling [24], bin packing [27], course timetabling [16]). They use evolutionary algorithms to generate problem instances [43, 42], whereas we take a constraint-based approach. Part of their work is analysing existing instances in benchmark suites and visualising the hardness distribution of instances for particular problems; our framework can be fruitfully combined with their detailed analysis and visualisation methods.

Instance generators have been applied to hard problems in Operations Research as well. For example, NSPLib [48] provides an instance generator and large sets of nurse rostering instances. Their instance generator characterizes an instance through various complexity indicators, including problem sizes, preference distribution measures, coverage distribution measures, and time related constraints. They implement a dedicated procedure for generating instances with properties corresponding to the values of specific indicators as parameters. For the knapsack problem, [37] uses instance generators to identify the regions of the instance space that contain difficult instances. For the traveling thieves problem, [14] uses instance generators that discriminate between more than two options simultaneously.

In communities such as SAT, there have been various works [41, 21] that try to address the generation of instances with desired properties. The SAT competition [19] organisers partly crowdsource the creation of the evaluation set. They require participants to send 20 new instances each, guaranteeing that the competition is run on instances mostly unseen to the solver developers prior to the competition. In addition, a set of previously used instances is manually and carefully selected, using various criteria such as hardness and variety.

The problem of generating a good set of benchmark instances is also studied in the AI planning community [45]. SMAC [23], a tool for optimizing algorithm parameters, is paired

**Figure 2** An illustration of irace's tuning process.

with hand-coded programs to generate many sets of instances that smoothly scale in difficulty. Afterwards, a subset of the generated sets is selected, according to various criteria such as difficulty and fairness. This results in a set of instances that better reflect the differences between planners when compared to the instances used in the competition.

A related field of study is algorithm configuration/selection, including portfolio-based approaches (SATZilla [49, 50], CPHydra [36], sunny-CP [8, 28]). For these purposes it is important to have a sufficient number of instances with a variety of difficulties that can discriminate between the options [39].

## 3 Constraint-based Automated Instance Generation

Following the approaches in [4] and [3], our instance generation system AutoIG makes use of the essence constraint modelling pipeline [1] and the automated algorithm configurator irace [29]. The system receives as input a problem description model, a parameterised instance generator written as a constraint model (referred to as the *generator model*), the solver(s) for which we want to generate graded or discriminating instances, and the types of instances we are interested in (SAT or UNSAT or both). The role of the essence pipeline is to express the generator model and to create *candidate instances* by solving instances of the generator model (referred to as *generator instances*), while the role of irace is to search in the parameter space of the generator model, or in other words, to sample in the generator instance space, to find configurations that can give us candidate instances with the desired properties. In this section, we first describe the search procedure of irace (Section 3.1). We then explain how irace and constraint modelling are combined in the instance generation process of AutoIG (Section 3.2). Finally, we discuss in detail how each candidate instance is evaluated during AutoIG search using gradedness or discriminating criteria (Section 3.3).

### 3.1 irace's Tuning Process

irace [29] is a general-purpose automated algorithm configuration tool for finding the best configurations of a parameterised algorithm. One of its key ideas is *racing* [30]: using statistical tests to eliminate poor configurations early, avoiding wasting computational budget on less promising areas of the configuration space. irace leverages this idea with an iterated procedure where each iteration is a *race* among several configurations. Figure 2 illustrates irace's tuning process. At the first iteration, a number of random configurations are generated, and a race started by evaluating all configurations on a subset of a given instance set, on a number of random seeds if the algorithm studied is stochastic, or a combination of both. A statistical test is applied to identify and eliminate the worst configurations.

**Listing 1** A fragment of an example for racp problem

```
1  % --- Fragment of MiniZinc model (succ: the immediate successors of tasks) --
2  array [int(1..n_tasks)] of set of int(1..n_tasks): succ;
3  % --- Fragment of generator model, in Essence ---
4  given n_tasks_t : int(1..60)         given s_density : int(1..5)
5  find succ: matrix indexed by [int(1..n_tasks_t)] of set of int(2..n_tasks_t)
6  such that sum([ |succ[t]| | t : int(1..n_tasks_t) ])/n_tasks_t = s_density
7  % --- Fragment of an example generator instance, in Essence ---
8  letting n_tasks_t = 6        letting s_density = 2
9  % --- Fragment of an example candidate instance, in MiniZinc ---
10 succ = [{2, 4, 5, 6}, {3, 4, 5}, {4, 5, 6}, {6}, {6}, {}];
```

Evaluation proceeds with the remaining configurations and a statistical test is conducted again. This is repeated until only a few good configurations remain or when the budget for the current race has been used. The race is then finished and the surviving configurations are used to update a sampling model. In the next iteration, new configurations are generated based on the updated sampling model and a new race is started. Tuning terminates when a given number of evaluations is exhausted, and the best configuration(s) recorded are returned.

## 3.2 AutoIG's Instance Generation Process

We give an example of the instance generation process in Listing 1, based on racp (see Section 4 for details). Fragments of a problem description model, a generator model, a generator instance, and a candidate instance are shown. In this example, a parameter (`succ`) of the problem description model (line 2) is written as a decision variable in the generator model (line 5). The creation of `succ` is controlled by tunable integer parameters of the generator model: `n_tasks_t` (equivalent to `n_tasks` in the original problem description); and `s_density`. Given an instance of the generator model sampled by irace (line 8), a candidate instance (line 10) can be created by solving the generator instance.

AutoIG utilises irace for searching in the configuration space of the generator model. The instance generation process starts with irace creating a number of random *generator configurations* (a configuration is an instance of the generator model, or in short, a *generator instance*). Each configuration is then evaluated using the procedure described in Algorithm 1 and a penalty is given back to irace for the statistical test. The tuning of irace then proceeds as normal, interleaving using constraint solving to generate new instances and to evaluate them, and using feedback from the evaluation process to eliminate non-promising configurations and to update the sampling model.

During each configuration evaluation, the generator instance $G$ is first solved via the essence pipeline (line 3 of Algorithm 1), whose solving procedure includes two translation steps by the automated constraint modelling tool conjure [6, 5] and by savilerow followed by a call to the constraint solver minion [20]. If $G$ is unsatisfiable or if it is too large to go through the pipeline, a very large penalty is returned so that irace will remove the configuration from the current race immediately (line 5). If $G$ is not solved by minion within the current evaluation, a penalty of 1 is returned. Otherwise, the new candidate instance $I$ is added to the solution history of $G$ to ensure that in the subsequent evaluations of this configuration, the same instance will not be generated again. Solution history is implemented via adding a negative constraint table into the minion input of $G$, and this table is constantly updated every time $G$ is evaluated during the tuning. Finally, the candidate instance $I$ is evaluated using one of the two instance evaluation procedures described in Algorithm 2 (for

**Algorithm 1** An evaluation of a generator configuration

1: **Input**: generator model $M$, generator instance $G$, solution history $H_G$
2: **Output**: penalty $p$
3: $r$ $solve(M, G, H_G)$ solve the generator instance $G$ using the essence pipeline
4: **if** $r$ is either UNSAT or timeout on savilerow **then**
5: **return** + return a very large penalty, irace will discard $G$ immediately
6: **if** $r$ is timeout on minion **then**
7: **return** 1
8: $I$ the instance generated by $r$
9: Add $I$ into $H_G$
10: $p$ Evaluate $I$ using either GRADED or DISCRIMINATING procedure
11: **return** $p$

**Algorithm 2** An evaluation of an instance using gradedness criteria

1: **Input**: problem specification $P$, instance $I$, solver $S$, minimum solving time $t_{min}$, maximum solving time $t_{max}$, instance types $T$ (that we are interested in)
2: **Output**: penalty $p$
3: **procedure** Graded($P, I, S, t_{min}, t_{max}, T$)
4: $r$ $solve(P, I, S, t_{max})$ solve $I$ using $S$ with time limit $t_{max}$, save results to $r$
5: **if** $solving_time(r) < t_{min}$ or $r$ is timeout **then**
6: **return** 0 $I$ is either too easy or too difficult for $S$
7: **if** $instance_type(r)$ $T$ **then**
8: **return** 0 $I$ is not the instance type we are interested in
9: **return** -1

graded instance generation) or Algorithm 3 (for discriminating instance generation), and the corresponding penalty is returned to irace. Note that the default setting of irace uses the Friedman test, a rank-based statistical test. This is also the setting used by AutoIG, i.e., the magnitude of difference in the penalty values between evaluations is not taken into account, only the rankings between them matter.

## 3.3 Evaluating Graded and Discriminating Instances

AutoIG's instance generation process depends heavily on an effective way of evaluating the quality of candidate instances. In this section, we describe the algorithms used for evaluating whether each candidate instance is graded or for measuring their discriminating power. The algorithms given in this section are invoked in line 10 of Algorithm 1.

To evaluate whether a candidate instance is graded, we employ Algorithm 2. This algorithm has 6 inputs: a problem specification $P$ of the problem under study, an instance $I$ and a solver $S$ to be evaluated, the range of solving times ($t_{min}$ and $t_{max}$) for the instance to be considered graded for $S$ (to avoid instances that are too easy or too hard to solve), and the type of instances ($T$) that we are interested in (either satisfiable, unsatisfiable, or both). The instance is first solved by $S$ (line 4) (See Algorithm 2). Results of the solving ($r$) include the status of the solving process (timeout/UNSAT/SAT), and the returned solution $I$ (if status is SAT). In our experiments $S$ is called via the MiniZinc toolchain. For complete solvers, we use the amount of time to solve the instance to completion (i.e., with a claim of optimality for optimisation problems, or with a feasible solution returned for decision

**Algorithm 3** An evaluation of an instance using discriminating criteria

1: **Input**: problem specification $P$, instance $I$, favoured solver $S_F$, base solver $S_B$, minimum solving time $t_{min}$ (for $B$ only), maximum solving time $t_{max}$, instance types $T$
2: **Output**: penalty $p$
3: **procedure** Discriminating($P, I, S_F, S_B, t_{min}, t_{max}, T$)
4: $r_F$ $solve(P, I, S_F, t_{max})$ solve $I$ using $S_F$ with time limit $t_{max}$
5: $r_B$ $solve(P, I, S_B, t_{max})$ solve $I$ using $S_B$ with time limit $t_{max}$
6: **if** $r_F$ is timeout or $instance_type(r_F)$ $T$ or $solving_time(r_B) < t_{min}$ **then**
7: **return** 0 $I$ is either too difficult for $S_F$, or not the right instance type, or too easy for $S_B$
8: $score_F, score_B$ MiniZinc_Score($S_F, S_B, P, I$)
9: **if** $score_F = 0$ and $score_B = 0$ **then**
10: **return** 0
11: **return** $-score_F / score_B$ When $score_B = 0$, returns large negative number.

problems or a claim of unsatisfiablity). For local search solvers such as Yuck, since a proof of optimality cannot be achieved for optimisation problems, we use an external complete solver (called the "oracle") to solve the instance to optimality (with a much longer time limit than $t_{max}$), and use that to measure the time until $S$ first finds the optimal solution. If the instance turns out to be too easy for $S$ or if the solving process times out (line 5) or the instance type is not interesting to the users (line 7), a penalty of 0 is given back to irace. Otherwise, the instance is considered graded and a negative penalty of $-1$ is returned.

Algorithm 3 is used for evaluating the discriminating power of an instance between two solvers. Each evaluation requires two input solvers: a *favoured solver* $S_F$ and a *base solver* $S_B$. We want to find instances that are easy to solve by $S_F$, while being difficult for $S_B$. The idea is to measure the performance of both solvers on the same instance, and search for instances that maximise the difference in performance. To avoid cases where the performance difference may be due to time measurement sensitivity, we impose a minimum solving time $t_{min}$ on the base solver $S_B$, i.e., the discriminating instances must be non-trivial to solve by $S_B$. Similar to the gradedness evaluation, AutoIG also allows focusing on a particular instance type during the generation process.

The evaluation of the discriminating property starts by applying $S_F$ and $S_B$ on the given instance (lines 4 and 5, Algorithm 3). If the instance does not satisfy our acceptance conditions (incorrect type, too easy for the base solver $S_B$ or unsolvable by the favoured solver $S_F$ (line 6)) a penalty of 0 is returned. Otherwise, we calculate the discriminating power of the instance and use it as feedback to irace. The discriminating power is calculated as the ratio between the performance of the favoured solver and the base solver, and the aim of the tuning process is to maximise this ratio. To take into account both solving time and solution quality when evaluating the performance of a solver, we use the *complete scoring* approach of the MiniZinc competitions. After calculating the MiniZinc scores of both solvers (line 8), the discriminating score is calculated as the MiniZinc score of $S_F$ divided by the MiniZinc score of $S_B$ and the negation of that ratio is returned to irace (line 11). Note that when both MiniZinc scores are equal to 0, the discriminating score is set to 0 (line 10).

The MiniZinc (complete) score for calculating the relative performance of two solvers on an instance can be found on the competition website (`https://www.minizinc.org/challenge2021/rules2021.html#assessment`). For completeness, in the rest of this section we will describe this score calculation in detail.

**Algorithm 4** Check whether one solver performs better than another in terms of solution quality

1: **Input:** solver $A$, solver $B$, problem model $P$, instance $I$
2: **procedure** IsBetter($A, B, P, I$)
3:     **if** $P$ is a decision problem **then**
4:         **return** solved($A, P, I$) and not solved($B, P, I$)
5:     **else**
6:         **return** (solved($A, P, I$) and not solved($B, P, I$)) or
7:         (optimal ($A, P, I$) and not optimal ($B, P, I$)) or
8:         (quality($A, P, I$) is better than quality($B, P, I$))

**Algorithm 5** MiniZinc score calculation between two solvers.

1: **Input:** solver $A$, solver $B$, problem model $P$, instance $I$
2: **procedure** MiniZinc_Score($A, B, P, I$)
3:     **if** IsBetter($A, B, P, I$) **then**
4:         $score_A$ 1, $score_B$ 0
5:     **else if** IsBetter($B, A, P, I$) **then**
6:         $score_A$ 0, $score_B$ 1
7:     **else if** solved($A, B, P, I$) **then**
8:         $score_A$ time($B, P, I$)/(time($A, P, I$)+time($B, P, I$))
9:         $score_B$ $1 - score_A$
10:     **else**
11:         $score_A$ $score_B$ 0
12:     **return** $score_A$ and $score_B$

Given a solver $S$, a problem model $P$ and an instance $I$, the following information is collected for the calculation: time($S, P, I$) – the solving time of $S$ on $I$; solved($S, P, I$) – whether a correct solution or a correct unsatisfiability result for $I$ is returned by $S$; quality($S, P, I$) – the best objective value obtained by $S$; and optimal ($S, P, I$) – whether a claim of optimality is returned by $S$. Based on those information, the function IsBetter($A, B, P, I$) (Algorithm 4) determines whether solver $A$ is clearly better than solver $B$ in terms of solution quality, for decision problems (line 4) and for optimisation problems (lines 6-8).

Finally, the MiniZinc complete score when comparing two solvers on an instance $I$ is calculated in Algorithm 5. The calculation starts with checking whether one of the two solvers is better than the other in term of solution quality (lines 3-6). If that is not the case, there are two possibilities. First, $I$ is solved by both solvers, and for optimisation problems, the same solution quality is achieved by both. In that case the normalised solving times are used as the scores. Second, both solvers fail to solve $I$, and in that case a score of 0 is returned for both. Note that this is slightly different from the scoring used in the MiniZinc competitions, where the scores of 1 and 0 are given to $A$ and $B$, respectively. This is because the final competition ranking is based on the Borda counting system, where the score is calculated for all pairs of solvers, including the same pair in the opposite order.

# 4 Case Studies

In this section we describe the five problems that are used to evaluate AutoIG, and also the set of four solvers that are used in our experiments. The five problems being used in this study are taken from the latest MiniZinc Challenges. They are chosen with the aim

of covering a variety of different problem properties, including the existence of redundant and symmetry breaking constraints, the usage of different global constraints, and a range of problem domains. In this section, we give a brief overview of those problems and how their instance generation problems are modelled.

Multi-Agent Collaborative Construction problem (macc) [26]: This is a planning problem that involves constructing a building by placing blocks in a 3D map using multiple identical agents. Ramps must be built to access the higher levels of the building. The objective is to minimise the makespan (primary) and the total cost (secondary).

In addition to the basic parameters of a macc instance indicated in the problem specification (i.e., the number of agents, the time horizon and the map sizes), the instance generation process should include information about the building itself as this is likely to affect instance difficulty. Therefore, two parameters and related constraints are added to the generator model to represent the density of the building on the ground level and its average height.

Carpet Cutting problem (carpet-cutting): The Carpet Cutting Problem [40] is a packing problem in which room and stair carpets composed of rectangular sections must be packed onto a carpet roll of fixed width and whose length must be minimised. The problem is complicated by the ability to rotate the carpets to aid in the packing process.

This problem requires substantial instance data, including the specification of the constituent rectangles of each carpet, their dimensions, and the permitted carpet rotations. There are several implicit constraints on this data that are not captured in the original MiniZinc model and hence these must be injected into the instance generation process through our generator specification. In particular, the rectangles that comprise a carpet must not overlap and must form a contiguous shape, as well as have bounded sizes so as to avoid trivially unsatisfiable instances.

Mario problem (mario): The Maximum Profit Subpath Problem is a routing problem that requires us to find a path in a graph where the path endpoints are given. This path is subject to two main constraints, where the sum of weights associated to arcs in the path is restricted (fuel consumption), while the sum of weights associated to nodes in the path has to be maximized (reward).

Regarding the instance generation process, in addition to the basic parameters, the amount of reward per node is represented as a non-negative integer array, while the non-negative cost for each arc is represented as a 2-dimensional matrix. There are a few implicit constraints not represented in the MiniZinc model, where the initial and goal nodes are different and have 0 reward, and the cost matrix is symmetric on the diagonal.

Resource Availability Cost Problem (racp): The Resource Availability Cost Problem [25] is a scheduling problem with activities that are non-interruptible and have a fixed duration. The problem includes precedence constraints between pairs of activities $i, j$ (that require activity $i$ to be completed before activity $j$ begins), arranged in a directed acyclic graph. There are a set of renewable resources, and each activity (when running) requires a given amount of each resource. All activities must be completed by a given deadline. Each resource has a cost per unit, and the objective is to minimise the peak costs of the resources.

The durations of activities, unit costs of resources, and resource demands of activities are all matrices of integers without complex constraints. However, the precedence graph (represented as a set of successors for each activity) has implicit constraints that are not represented in the MiniZinc model. Firstly, it must be acyclic, and we achieve this by mapping activities to numbered layers and allowing only edges from lower to higher-numbered layers. Secondly, we ensure that each activity has at least one predecessor and at least one successor (except the dummy first and last activities).

Discrete Lot Sizing problem (lot-sizing): The Discrete Lot Sizing and Scheduling Problem [22, 46] (CSPLib 58) requires us to find a production schedule for a set of orders, each with a due date within a planning horizon. There are various costs associated with production, such as setup, changeover and stocking costs, the sum of which must be minimised.

This problem requires substantial instance data including the type and due date of each order, and moreover a table of changeover costs between orders. There are a number of implicit constraints on this data, including a dummy order type 0 which incurs 0 cost to change to/from, and the fact that the changeover costs for the remaining order must obey the triangle inequality. Again, these are not captured in the original MiniZinc model and hence must be injected into the instance generation process through our generator specification.

We investigate the performance of four solvers, also taken from the MiniZinc challenges, on the problems described above using our framework. They are chosen such that a variety of solving techniques and different competition rankings are included. The solvers are: OR-Tools [2] (version 9.2) – a systematic solver from Google that combines CP, SAT, and linear programming techniques; Picat-SAT [51] – a SAT compiler for the multi-paradigm programming language Picat which uses kissat [12] as the underlying SAT solver; Chuffed [15] (version 0.10.4) – a clause learning CP solver which was not a participant of the challenges but was used in the score calculation process to rank participating solvers; and Yuck [31] (version 20210501) – a constraint-based local search solver.

OR-Tools has consistently won the last several competitions and Picat-SAT has received multiple silver medals. Yuck is the winning solver in the Local Search category of the 2020 and 2021 competitions. However, its ranking was generally low when compared to OR-Tools and Picat-SAT. In particular, based on the competition data, it was completely dominated by OR-Tools on the five problems considered.

## 5 Experimental Setup

The first set of experiments are on generating graded instances. For each problem, we first generate graded instances for each solver via an AutoIG experiment with a budget of $2,000$ runs. Note that a run is an evaluation of a generator configuration. The gradedness criteria is defined as being solvable by the given solver with the time ranging from 10 seconds (to avoid trivial instances) to 20 minutes (the time limit used by the MiniZinc Challenge). Following the competition approach, MiniZinc translation time is included in the total time measured. Since Yuck is a local search solver, we use OR-Tools (with a budget of 1 hour) for checking whether a solution returned by Yuck is optimal. After all graded instances are collected, we then randomly select 50 graded instances from each experiment to get a combined benchmark instance set for each problem. Finally, we evaluate the performance of all four solvers on the combined instance set.

The second set of experiments are on generating discriminating instances. Since OR-Tools has consistently shown very strong performance on the competition data, the main aim of these experiments is to see whether we can find instances where OR-Tools is performing worse than the other two participating solvers being considered. We do this without loss of generality: our discriminating instance generation procedure can be applied to any pair of solvers. We compare two solvers (Picat-SAT and Yuck) against OR-Tools. For each solver we conduct two separate AutoIG experiments, one where we search for instances that are solved more quickly by OR-Tools and one for the opposite case. The same AutoIG budget and memory limit as in graded experiments are used. To avoid instances where the difference between the performance of two solvers is due to fluctuations in running time measurement,

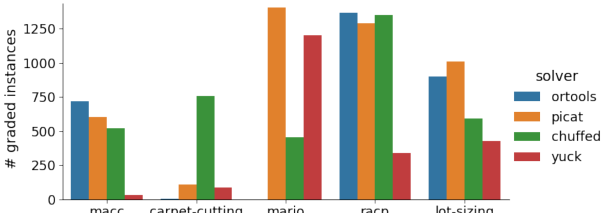


**Figure 3** Number of graded instances generated.

a minimum requirement of 10 seconds is imposed on the solving time of the base solver, i.e. instances that can be trivially solved by the base solver are discarded.

All experiments were performed on a computing node of a High Performance Computing cluster. Each node is equipped with two 2.1 GHz, 18-core Intel Xeon processors and 256 GB RAM. Each solver except Yuck is given a memory limit of 8GB via the runsolver tool [38]. For Yuck, the memory limit is controlled directly via the Java Runtime Environment (JRE). For solving the generator models, time limits of 5 and 10 minutes are given to savilerow and minion, respectively. In this work, we focus on the Free Track of the competitions. Therefore, all solvers are called via the MiniZinc toolchain with a single core and with the free search option being passed to the solver. Although AutoIG supports focusing on generating either only SAT or only UNSAT instances, in this work we allow both types of instances to be generated.

## 6 Results on graded instances

First we describe the sets of graded instances produced by AutoIG for the five problems (Section 6.1) and discuss insights obtained from analysing the results. Then in Section 6.2 we combine the sets of graded instances for each problem, and re-evaluate the four solvers using the combined sets of instances, showing substantially different relative performance in some cases compared with the competition instances.

### 6.1 Graded instance generation

For each problem, Figure 3 shows the number of graded instances obtained per solver within the given budget. While we can achieve more than a few hundred graded instances in most cases, there are cases where we are only able to generate a small number of instances. For example, with OR-Tools on carpet-cutting and mario, we generate only 4 and 1 graded instances, respectively. In addition, the numbers are fairly small for Yuck on macc and carpet-cutting. There is a large variation in the number of graded instances we are able to generate for different problems and solvers (shown in Figure 3).

The differences in the number of graded instances returned by each experiment suggest that the performance of the solvers varies significantly when solving instances drawn from the same instance space. In order to better understand the performance distribution of each solver we investigate the details of the search space of AutoIG. More specifically, we check the status of each configuration evaluation run and measure their frequency, as detailed in Figure 4. For OR-Tools on carpet-cutting and mario, only a small number of graded instances are found, but this same outcome has entirely different causes. For carpet-cutting, almost half of the runs are with unsolvable generator configurations, and for the rest the candidate instances are mostly trivially proved unsatisfiable by OR-Tools.

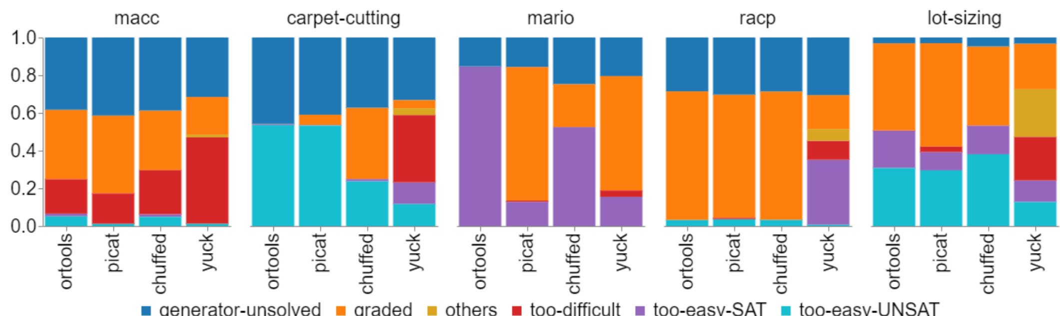


**Figure 4** Frequency of all run statuses, including `generator-unsolved` (generator instance is UNSAT or unsolvable); `graded` (a graded instance is obtained); `too-difficult` (the candidate instance is unsolvable by the considered solver within the time limit); `too-easy-SAT` and `too-easy-UNSAT` (the candidate instance is too easy, i.e., solved within less than 10 seconds); and `others` (the considered solver fails due to unexpected errors such as incorrect returned answers).

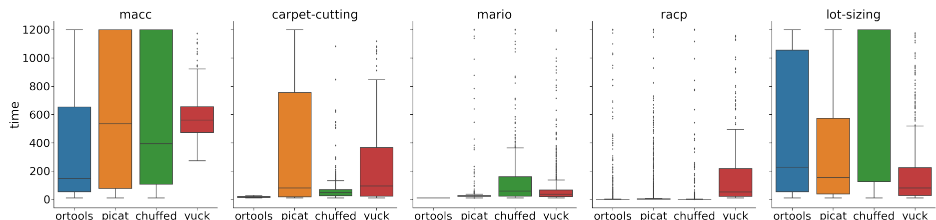


**Figure 5** Solving time of graded instances generated for each pair of problems and solvers. Note that the instances presented here are the graded instances found for each solver independently. The performance of these solvers on the combined set of graded instances can be seen in Figure 6.

For mario, the majority of the runs produce instances that are trivially satisfiable. Once we understand the underlying reason for the lack of graded instances, we can rectify each of these shortcomings: for carpet-cutting, expert knowledge on the problem may be added as constraints to the generator model to avoid trivially unsatisfiable instances, while for mario, the current instance space may be too easy for OR-Tools and we may want to increase the upper bounds of some of the generator parameters. On the other hand, the situation is completely different for Yuck: the small number of graded instances obtained for macc and carpet-cutting is largely due to the fact that the majority of instances generated are too difficult to solve.

In addition to the run statuses, the distribution of solving time of graded instances also gives us interesting insights into the performance of different solvers, as illustrated in Figure 5. Notably, many graded instances for mario and racp are close to the lower bound of graded instances; this is true for all solvers. Nevertheless, AutoIG is able to find challenging graded instances, which can take several hundred seconds to solve, for all solvers on those two problems (except for OR-Tools on mario). For carpet-cutting, OR-Tools and Chuffed can solve most graded instances quickly, while Picat-SAT and Yuck take more time in general. Finally, for macc and lot-sizing, the solving time distributions of all four solvers are more well-spread, indicating a good diversity of difficulties among the generated graded instances.

Note that for the majority of graded instances generated, the MiniZinc flattening times are generally marginal compared to the time taken to solve them. This indicates that the more difficult graded instances are actually challenging for the solvers themselves, and can be useful for solver developers to improve their solver performance.

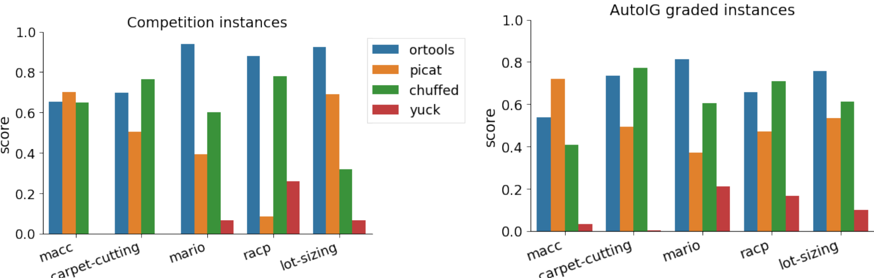


**Figure 6** MiniZinc Borda (complete) scores of each solver on the MiniZinc Challenges instance set (left) and on the combined graded instance set generated by AutoIG (right).

## 6.2 Comparison of Solver Performance on Graded Instances

We combine all graded instances to construct a diverse set of instances for each problem. We then evaluate all four solvers on the combined set and rank them using the Borda (complete) scoring method of the MiniZinc Challenge (`https://www.minizinc.org/challenge2021/rules2021.html`). More specifically, for each problem, 50 graded instances are uniformly sampled from the set of graded instances for each solver. In cases where there are less than 50 graded instances available, we just take them all. For comparison, we also evaluate those solvers on the instances used in the competition. There are 5-10 instances per problem, as some problems are re-used over two different competitions.

Figure 6 shows the scores on the competition instances (left) and on the combined graded instances generated by AutoIG (right). There are similarities between results on the two sets of instances. Performance of OR-Tools and Chuffed remain strong in most cases, followed by Picat-SAT. For macc, carpet-cutting and mario, the overall rankings of the four solvers on both groups are almost the same. However, results on the graded set do show certain changes in relative performance of all solvers. For example, the scores of Yuck on the graded instances are no longer zero for macc and carpet-cutting, and the score for mario increases noticeably. This indicates that Yuck is actually not completely dominated by all other solvers on those three problems as suggested by the competition data. For racp, the ranking has changed significantly: OR-Tools swaps places with Chuffed, and Picat-SAT swaps places with Yuck. For lot-sizing, Picat-SAT is no longer ranked higher than Chuffed.

Thanks to the solution checking process being integrated into each evaluation, we also found a number of cases from the combined graded sets where incorrect answers are returned, which can be of separate interest to the solver developers. There were 41 (out of 183) macc instances and 90 (out of 154) carpet-cutting instances (from the subset of graded instances generated for other solvers) where Yuck reports objective values of infeasible solutions.

Generating a larger number of graded instances for each solver and analysing them using the presented methods gives more information in comparison to a typical competition's result, which would be a ranking of the solvers. In Section 7 we apply the discriminating instance generation feature of AutoIG to gain even more insight into solver performance.

# 7 Results on Discriminating Instances

Results on MiniZinc competition data indicate that OR-Tools is a very strongly performing solver on the 5 problems considered. It completely dominates Yuck, i.e., Yuck gets zero score on all competition instances when compared directly to OR-Tools. OR-Tools also wins over Picat-SAT on all instances of mario and racp, on 9 out of 10 instances of lot-sizing, and

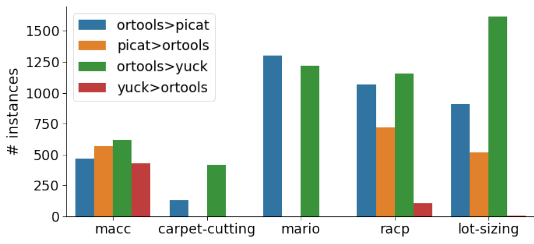


**Figure 7** Number of discriminating instances generated per *favoured* and *base* solver pair.

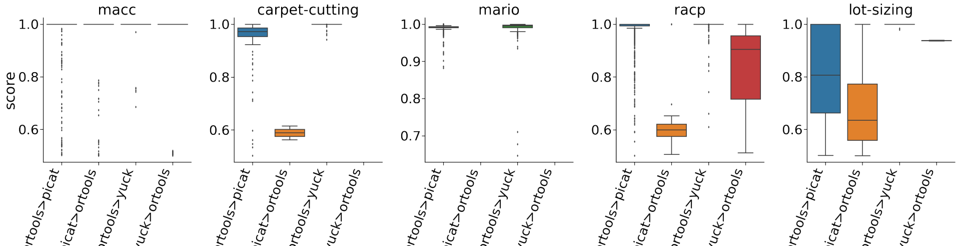


**Figure 8** Distribution of scores (of the winning solver) on discriminating instances generated.

on 8 out of 10 instances of carpet-cutting. However, detailed results obtained from the evaluation on graded instances suggest that this may not always be the case. For example, there are 31 instances evaluated on racp where Picat-SAT performs better than OR-Tools, and 58 macc instances where Yuck performs better. In this section, we use the discriminating instance generation feature of AutoIG to get more insights into these cases.

Figure 7 shows the number of discriminating instances generated for the two pairs of solvers. In the experiments on OR-Tools versus Yuck, AutoIG found 431 macc instances and 110 racp instances where Yuck gets a better score than OR-Tools, which indicates that Yuck is not completely dominated by OR-Tools on these two problems. On the other hand, for carpet-cutting and mario, results suggest that Yuck may indeed be entirely dominated by OR-Tools, as no instances were found in the experiments that favour Yuck. Furthermore, for lot-sizing, only 3 discriminating instances favouring Yuck are found. In the experiment on OR-Tools versus Picat-SAT, OR-Tools shows domination on both carpet-cutting (only 2 instances where Picat-SAT is better than OR-Tools were found) and mario (no instances favouring Picat-SAT was found). On the other three problems, there are a good number of discriminating instances in both directions.

The number of discriminating instances tell us if winning instances for a solver can be found, but it does not show the magnitude of the difference in performance. We can get additional insights into comparative performance of the solvers by looking into the detailed scores of the winning solver on discriminating instances for each experiment. As shown in Figure 8, for macc, the median lines indicate that for all four cases, several discriminating instances found have the highest "discriminating power", i.e., the winning solver gets the maximum score of 1 (the other solver, in turn, gets zero score). This type of instance is probably the most interesting for understanding the shortcomings of a particular solver. For carpet-cutting, on the only 2 discriminating instances where Picat-SAT has better score than OR-Tools, the score distribution of the corresponding experiment (Picat-SAT>OR-Tools) suggests that OR-Tools performance is not much worse. This suggests that OR-Tools indeed dominates Picat-SAT on this problem. A similar conclusion can be reached for Yuck,

i.e., it is clear that OR-Tools is really the dominating solver on carpet-cutting since the magnitude of the performance difference is very small even for the instances where Yuck is faster. Similarly, for mario, OR-Tools very clearly dominates in comparison to Picat-SAT and Yuck, as indicated by the discriminating score distributions. This is in line with what was observed in the previous section's results on the same problem.

Interestingly, for racp, although the number of discriminating instances of Picat-SAT>OR-Tools is larger than of Yuck>OR-Tools as shown in Figure 7, the magnitude of the performance difference of instances found for Yuck is generally much higher. This observation gives a new insight that has not been revealed in all previous experiments on gradedness: even though the performance of Yuck is dominated by other solvers in general (i.e., it is ranked lower) and it has a smaller number of discriminating instances favouring it, the magnitude of the performance difference is very large for these instances. This means there exists a subset of the racp instances where Yuck's performance is much better than OR-Tools, while this does not seem to be the case for Picat-SAT.

The insights provided by discriminating instances could be useful in constructing a robust portfolio of solvers for a given problem. For example, on racp, Yuck is the weakest solver by a wide margin on the graded instances (see Figure 6) and second-weakest on competition instances. On the graded instances, Picat-SAT performs considerably better than Yuck. However, the results with discriminating instances show that Yuck would be a good candidate to add to a portfolio (alongside OR-Tools) whereas Picat-SAT may not be.

## 8 Conclusions and Future Work

Assessing the performance of solving methods via benchmark problems is fundamental to CP research. However, its utility is limited by the availability of problem instances that are of suitable difficulty, and diverse (not inadvertently favouring one solver over another). We have shown that our system AutoIG can generate large numbers of informative benchmark instances graded for difficulty for a single solver, or that can discriminate between two solvers (favouring one or the other). The only manual part of the AutoIG process is to capture (in a generator model) any implicit constraints on the instances data.

The essential task of benchmarking is to compare multiple solvers and rank them. As illustrated in our experiments, AutoIG can be used to generate graded instances for each solver independently, and these can then be combined into one set of instances, providing confidence that the generation process does not favour one solver or class of solvers. Furthermore, we have shown that automatically generated instances can provide more detailed insights than just a ranking. Instances generated by AutoIG can reveal cases where a solver is weak or even faulty, providing valuable information to solver developers. Finally, discriminating instances can reveal parts of the instance space where a generally weak solver performs well relative to others, and therefore could be useful as part of a portfolio.

There are various directions for future improvement. First, the diversity of instances found during search can be taken into account to increase the quality of the final instance set. This would require a definition of diversity, which could be based on problem-specific instance features or on general constraint programming features such as the fzn2feat features [7]. Secondly, similar to the series of work on Instance Space Analysis (e.g. [32, 24, 16]), a detailed visualisation of the instance space based on performance data collected from the tuning and evaluation process of AutoIG would provide further insights into performance of the solvers under study. Again, instance features would be needed for such analysis.

## References


1 essence modelling pipeline:. https://constraintmodelling.org/.
2 Google OR-Tools, 2021. Available from https://github.com/google/or-tools.
3 Özgür Akgün, Nguyen Dang, Ian Miguel, András Z Salamon, Patrick Spracklen, and Christopher Stone. Discriminating instance generation from abstract specifications: A case study with CP and MIP. In *International Conference on Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, pages 41–51. Springer, 2020.
4 Özgür Akgün, Nguyen Dang, Ian Miguel, András Z Salamon, and Christopher Stone. Instance generation via generator instances. In *International Conference on Principles and Practice of Constraint Programming*, pages 3–19. Springer, 2019.
5 Ozgur Akgun, Ian P. Gent, Christopher Jefferson, Ian Miguel, and Peter Nightingale. Breaking conditional symmetry in automated constraint modelling with Conjure. In *Proceedings of the 21st European Conference on Artificial Intelligence (ECAI)*, pages 3–8, 2014.
6 Ozgur Akgun, Ian Miguel, Christopher Jefferson, Alan M Frisch, and Brahim Hnich. Extensible Automated Constraint Modelling. In Wolfram Burgard and Dan Roth, editors, *AAAI 2011 - Proceedings of the Twenty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2011, San Francisco, California, USA, August 7-11, 2011*. AAAI Press, 2011.
7 Roberto Amadini, Maurizio Gabbrielli, and Jacopo Mauro. An enhanced features extractor for a portfolio of constraint solvers. In *Proceedings of the 29th annual ACM symposium on applied computing*, pages 1357–1359, 2014.
8 Roberto Amadini, Maurizio Gabbrielli, and Jacopo Mauro. SUNNY-CP: a sequential CP portfolio solver. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, pages 1861–1867, 2015.
9 Clark Barrett and Cesare Tinelli. Satisfiability modulo theories. In *Handbook of model checking*, pages 305–343. Springer, 2018.
10 Thomas Bartz-Beielstein, Carola Doerr, Daan van den Berg, Jakob Bossek, Sowmya Chandrasekaran, Tome Eftimov, Andreas Fischbach, Pascal Kerschke, William La Cava, Manuel Lopez-Ibanez, et al. Benchmarking in optimization: Best practice and open issues. *arXiv preprint arXiv:2007.03488*, 2020.
11 Vahid Beiranvand, Warren Hare, and Yves Lucet. Best practices for comparing optimization algorithms. *Optimization and Engineering*, 18(4):815–848, 2017.
12 Armin Biere, Mathias Fleury, and Maximilian Heisinger. CaDiCaL, Kissat, Paracooba entering the SAT competition 2021. In T Balyo, N Froleyks, M Heule, M Iser, M Järvisalo, and M Suda, editors, *Proceedings of SAT Competition 2021: Solver and Benchmark Descriptions. Department of Computer Science Report Series B, vol. B-2021-1, Department of Computer Science, University of Helsinki, Helsinki*, 2021.
13 Armin Biere, Marijn Heule, and Hans van Maaren. *Handbook of satisfiability*, volume 185. IOS press, 2009.
14 Jakob Bossek and Markus Wagner. Generating instances with performance differences for more than just two algorithms. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, pages 1423–1432, 2021.
15 Geoffrey Chu, Peter J. Stuckey, Andreas Schutt, Thorsten Ehlers, Graeme Gange, and Kathryn Francis. Chuffed, 2018. Available from https://github.com/chuffed/chuffed/.
16 Arnaud De Coster, Nysret Musliu, Andrea Schaerf, Johannes Schoisswohl, and Kate Smith-Miles. Algorithm selection and instance space analysis for curriculum-based course timetabling. *Journal of Scheduling*, pages 1–24, 2021.
17 Luiz Henrique dos Santos Fernandes, Ana Carolina Lorena, and Kate Smith-Miles. Towards understanding clustering problems and algorithms: an instance space analysis. *Algorithms*, 14(3):95, 2021.
18 Nils Froleyks, Marijn Heule, Markus Iser, Matti Järvisalo, and Martin Suda. SAT competition 2020. *Artificial Intelligence*, 301:103572, 2021.

19 Nils Froleyks, Marijn Heule, Markus Iser, Matti Järvisalo, and Martin Suda. SAT competition 2020. *Artificial Intelligence*, 301:103572, 2021. doi:https://doi.org/10.1016/j.artint.2021.103572.

20 Ian P. Gent, Christopher Jefferson, and Ian Miguel. Minion: A fast scalable constraint solver. In *Proceedings ECAI 2006*, pages 98–102, 2006.

21 Jesús Giráldez-Cru and Jordi Levy. A modularity-based random SAT instances generator. In Qiang Yang and Michael J. Wooldridge, editors, *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI*, pages 1952–1958. AAAI Press, 2015. URL: http://ijcai.org/Abstract/15/277.

22 Vinasétan Ratheil Houndji, Pierre Schaus, Laurence Wolsey, and Yves Deville. The stockingcost constraint. In *International conference on principles and practice of constraint programming*, pages 382–397. Springer, 2014.

23 Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. Sequential model-based optimization for general algorithm configuration. In Carlos A. Coello Coello, editor, *Learning and Intelligent Optimization - 5th International Conference, LION 5, Rome, Italy*, volume 6683 of *Lecture Notes in Computer Science*, pages 507–523. Springer, 2011. doi:10.1007/978-3-642-25566-3_40.

24 Lucas Kletzander, Nysret Musliu, and Kate Smith-Miles. Instance space analysis for a personnel scheduling problem. *Annals of Mathematics and Artificial Intelligence*, 89(7):617–637, 2021.

25 Stefan Kreter, Andreas Schutt, Peter J Stuckey, and Jürgen Zimmermann. Mixed-integer linear programming and constraint programming formulations for solving resource availability cost problems. *European Journal of Operational Research*, 266(2):472–486, 2018.

26 Edward Lam, Peter J Stuckey, Sven Koenig, and TK Kumar. Exact approaches to the multi-agent collective construction problem. In *International Conference on Principles and Practice of Constraint Programming*, pages 743–758. Springer, 2020.

27 Kelvin Liu, Kate Smith-Miles, and Alysson Costa. *Using Instance Space Analysis to Study the Bin Packing Problem*. PhD thesis, 2020.

28 Tong Liu, Roberto Amadini, Maurizio Gabbrielli, and Jacopo Mauro. sunny-as2: Enhancing SUNNY for algorithm selection. *Journal of Artificial Intelligence Research*, 72:329–376, 2021.

29 Manuel López-Ibáñez, Jérémie Dubois-Lacoste, Leslie Pérez Cáceres, Mauro Birattari, and Thomas Stützle. The irace package: Iterated racing for automatic algorithm configuration. *Operations Research Perspectives*, 3:43–58, 2016.

30 Oden Maron and Andrew W Moore. The racing algorithm: Model selection for lazy learners. *Artificial Intelligence Review*, 11(1):193–225, 1997.

31 Michael Marte. Yuck, 2021. Available from https://github.com/informarte/yuck.

32 Mario A Muñoz, Laura Villanova, Davaatseren Baatar, and Kate Smith-Miles. Instance spaces for machine learning classification. *Machine Learning*, 107(1):109–147, 2018.

33 Mario Andrés Muñoz, Tao Yan, Matheus R Leal, Kate Smith-Miles, Ana Carolina Lorena, Gisele L Pappa, and Rômulo Madureira Rodrigues. An instance space analysis of regression problems. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 15(2):1–25, 2021.

34 Nicholas Nethercote, Peter J Stuckey, Ralph Becket, Sebastian Brand, Gregory J Duck, and Guido Tack. Minizinc: Towards a standard CP modelling language. In *International Conference on Principles and Practice of Constraint Programming*, pages 529–543. Springer, 2007.

35 Peter Nightingale, Özgür Akgün, Ian P Gent, Christopher Jefferson, Ian Miguel, and Patrick Spracklen. Automatically improving constraint models in Savile Row. *Artificial Intelligence*, 251:35–61, 2017.

36 Eoin O'Mahony, Emmanuel Hebrard, Alan Holland, Conor Nugent, and Barry O'Sullivan. Using case-based reasoning in an algorithm portfolio for constraint solving. In *Irish conference on artificial intelligence and cognitive science*, pages 210–216, 2008.

37 David Pisinger. Where are the hard knapsack problems? *Computers & Operations Research*, 32(9):2271–2284, 2005.

38 Olivier Roussel. Controlling a solver execution with the runsolver tool. *Journal on Satisfiability, Boolean Modeling and Computation*, 7(4):139–144, 2011.

39 Marius Schneider and Holger H Hoos. Quantifying homogeneity of instance sets for algorithm configuration. In *International Conference on Learning and Intelligent Optimization*, pages 190–204. Springer, 2012.

40 Andreas Schutt, Peter J Stuckey, and Andrew R Verden. Optimal carpet cutting. In *International Conference on Principles and Practice of Constraint Programming*, pages 69–84. Springer, 2011.

41 Bart Selman, David G. Mitchell, and Hector J. Levesque. Generating hard satisfiability problems. *Artificial Intelligence*, 81(1-2):17–29, 1996. doi:10.1016/0004-3702(95)00045-3.

42 Kate Smith-Miles, Jeffrey Christiansen, and Mario Andrés Muñoz. Revisiting where are the hard knapsack problems? via instance space analysis. *Computers & Operations Research*, 128:105184, 2021.

43 Kate Smith-Miles and Jano van Hemert. Discovering the suitability of optimisation algorithms by learning from evolved instances. *Annals of Mathematics and Artificial Intelligence*, 61(2):87–104, 2011.

44 Peter J Stuckey, Ralph Becket, and Julien Fischer. Philosophy of the MiniZinc challenge. *Constraints*, 15(3):307–316, 2010.

45 Alvaro Torralba, Jendrik Seipp, and Silvan Sievers. Automatic instance generation for classical planning. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 31, pages 376–384, 2021.

46 Hafiz Ullah and Sultana Parveen. A literature review on inventory lot sizing problems. *Global Journal of Research In Engineering*, 10(5), 2010.

47 Mauro Vallati, Lukás Chrpa, and Thomas Leo McCluskey. What you always wanted to know about the deterministic part of the international planning competition (IPC) 2014 (but were too afraid to ask). *Knowledge Engineering Review*, 33:e3, 2018. doi:10.1017/S0269888918000012.

48 Mario Vanhoucke and Broos Maenhout. On the characterization and generation of nurse scheduling problem instances. *European Journal of Operational Research*, 196(2):457–467, 2009.

49 Lin Xu, Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. SATzilla: portfolio-based algorithm selection for sat. *Journal of artificial intelligence research*, 32:565–606, 2008.

50 Lin Xu, Frank Hutter, Jonathan Shen, Holger H Hoos, and Kevin Leyton-Brown. SATzilla2012: Improved algorithm selection based on cost-sensitive classification models. *Proceedings of SAT Challenge*, 2012, 2012.

51 Neng-Fa Zhou and Håkan Kjellerstrand. Optimizing SAT encodings for arithmetic constraints. In *International Conference on Principles and Practice of Constraint Programming*, pages 671–686. Springer, 2017.