Discriminating Instance Generation from Abstract Specifications: A Case Study with CP and MIP

Özgür Akgün, Nguyen Dang, Ian Miguel, András Z. Salamon, Patrick Spracklen, and Christopher Stone

School of Computer Science, University of St Andrews, St Andrews, UK

Abstract. We extend automatic instance generation methods to allow cross-paradigm comparisons. We demonstrate that it is possible to completely automate the search for benchmark instances that help to discriminate between solvers. Our system starts from a high level human-provided problem specification, which is translated into a specification for valid instances. We use the automated algorithm configuration tool irace to search for instances, which are translated into inputs for both MIP and CP solvers by means of the Conjure, Savile Row, and MiniZinc tools. These instances are then solved by CPLEX and Chuffed, respectively. We constrain our search for instances by requiring them to exhibit a significant advantage for MIP over CP, or vice versa. Experimental results on four optimisation problem classes demonstrate the effectiveness of our method in identifying instances that highlight differences in performance of the two solvers.

Keywords: Instance generation · MIP · Constraint programming

1 Introduction

When developing a model of a combinatorial problem class, a set of representative instances drawn from the class is essential for evaluating the model's performance. Recently, Akgn et al. [1] demonstrated how to generate instances automatically from the Essence [1] specification of a problem class [10]. The instances generated are *graded*: neither too difficult nor too easy relative to a given solver and resource limits. Graded instances are particularly valuable for model evaluation, since they are less likely to be solved trivially with the model under development or to remain unsolved at the expiry of a time budget. Either of these outcomes would reveal little useful information about model performance.

In this paper we consider a more complex situation: rather than focus on modelling for a single solving paradigm, it is often the case that we might wish to evaluate two or more solving paradigms for a problem of interest. In this

[1] Essence is an abstract constraint specification language that supports a formal statement of a problem without committing to detailed modelling decisions.

context, it is desirable to generate instances that are not only graded, but also *discriminating*, i.e. which exhibit a pronounced difference in solving performance among the solving paradigms under consideration. Discriminating instances are valuable both for a manual inspection of the instance characteristics that favour one paradigm over the others, and to provide coverage of the instance space when training the selection process for an algorithm portfolio [28]. Our hypothesis is that starting from a single high-level specification of a problem we can generate discriminating instances automatically via synthesising an "instance generator model" and using standard algorithm tuning tools.

We consider two paradigms: Constraint Programming (CP) and Mixed Integer Programming (MIP). Extending the approach of Akgn et al., we employ the automated configuration tool irace [22] to search for graded discriminating instances for the CP solver Chuffed [7] and the MIP solver CPLEX [17]. This search is performed twice, with each solver in turn first considered the *base* solver, and the other the *favoured* solver. irace is guided to search for discriminating instances where the favoured solver performs significantly better than the base solver. The advantage of this approach is that, even when one of the two solvers predominantly performs better, the search for discriminating instances is pushed towards regions of the instance space where the generally weaker solver has the advantage. This provides good coverage of the instance space and a clear picture of relative solver performance, as our empirical results demonstrate.

2 Related work

Benchmarks play an important role in combinatorial optimisation as they are often the main device employed to verify the quality of solvers. For a long time this involved bundling a collection of problem instances, with one or more problem classes, that are then solved and compared by practitioners. In many cases, this has led to the reuse of the same set of instances for several decades causing algorithms to become highly tailored to solve those specific sets and becoming less generally applicable [13]. This is not an ideal practice, as it has been observed that different algorithmic techniques have their own strengths and weaknesses [30,6,19]. An alternative approach is making use of instance generators that can produce a stream of new instances. The two main approaches to generate instances in an automated manner are based on handcrafted programs [15,9,32], where practitioners use their knowledge to specify desired characteristics, and on meta-heuristic approaches where instances are created and selected according to some objective functions [29,18]. In both cases these generators can produce instances only for specific problem classes and making them applicable to new problem classes would require substantial modifications. Another criticism raised [31] is that often only a small number of algorithms are tested on a small set of instances to certify the superiority of one algorithm over another instead of studying the strengths and weaknesses of both.

Belov et al. [6] demonstrated automatic translation of CP specifications expressed in the MiniZinc language to lower-level FlatZinc [24], using knowledge of

the target paradigm to guide the translation. Their experiments showed that the MIP and CP solving paradigms have different sets of strengths and weaknesses, and found discriminating instances among the MiniZinc benchmarks.

Generation of discriminating instances is increasingly popular [4], for instance to measure algorithm performance across instances [28] or to improve algorithm selection tools [30,14]. Most studies in this area tackle one problem class at a time. We here extend [1] to automatically produce instances for any problem class. The goal is to automate finding instances that are particularly suited to one algorithm but not another, and to study characteristics of these instances.

3 Background

CP and MIP solvers work by solving a *problem instance* model composed of decision variables with associated domains, a set of constraints on the decision variables (and optionally an objective function for optimisation problems). CP modelling languages typically offer a richer language in comparison to MIP modelling languages thanks to having a richer set of constraint types. Modern modelling languages for both formalisms allow models to be written for a *problem class*. A problem class model is instantiated by a modelling tool before it is given to a solver to achieve a problem instance model.

Essence is a problem specification language for combinatorial decision and optimisation problems [10]. Essence supports *abstract* decision variables, such as multiset, relation and function, as well as *nested* types, such as multiset of sets. In addition to language features for specifying decision variables, constraints and the objective function (`find`, `such that`, `min/maximising` respectively) it allows the specification of problem parameters which define problem instances (`given`) and restrictions on values that parameters can take on valid instances (`where`).

Problem specifications written in Essence are converted to class level constraint models by Conjure [3,2], which are then fed into Savile Row [26] to instantiate the model and convert it into input suitable for a supported solver. Savile Row also applies instance level model improvements automatically [26].

A problem specification can be automatically converted to an *instance generator specification* by Conjure [1]. First, the `given` statements that declare parameters are converted into `find` statements that declare corresponding decision variables. Second, the `where` statements are converted to `such that` statements. This process is explained in more detail in [1].

The main objective of [1] was to generate *graded* instances (neither too easy nor too hard for a selected backend solver). This is achieved by using a general-purpose automated algorithm tuning tool to search for generator configurations covering the problem instance space, with solving time between the given bounds. In [1], irace was used for this task. irace is an automated algorithm configuration tool that supports tuning parameters of algorithms efficiently. The core idea behind irace is *iterated racing*, an iterative search procedure where at each iteration, statistical tests are used to eliminate configurations with poor performance early, so that the budget is saved for evaluating more promising configurations.

4 Experimental method

4.1 Problem classes

We demonstrate our method on the following four optimisation problem classes. The first three are typically solved by Operations Research methods. We work with existing Essence specifications from CSPlib [12], where available.

Transshipment (TP): Given costs of transporting goods from a warehouse to a transshipment point and from a transshipment point to a customer, warehouse stock levels, and customer demand, the objective is to minimise the total transport cost while meeting customer demand.TP is known to have efficient linear programming solutions [27] and we expected CPLEX to dominate Chuffed.

Progressive Party (PPP, CSPlib 013): The objective is to minimise the number of boats hosting a party at a yacht club, where some boats (with capacities) are designated as hosts, and the crews of the remaining boats visit the host boats for fixed time periods; two guest crews may meet at most once. PPP is a classic CP problem and we expected Chuffed to dominate CPLEX.

Warehouse Location (WLP, CSPlib 034): A central warehouse will supply depots, each with a maintenance cost and a capacity; each store will be supplied from exactly one depot at some cost. The objective is to find a subset of depots to open so as to minimise the sum of the maintenance and supply costs. We had no prior opinion on whether CP would outperform MIP.

Capacitated Vehicle Routing (CVRP): The task is to find least cost routes for identical vehicles with capacities, delivering goods from a central depot. Each location is visited once by one vehicle. A route starts at the depot and finishes there [8,20]. We had no prior opinion on whether MIP would outperform CP.

4.2 irace's scoring function to find discriminating instances

Each *evaluation* during the tuning involves a generator configuration and a random seed, both sampled by irace. The CP solver `minion` [11] is used to solve the configuration with the given random seed. A solution is returned as an instance of the original problem. That instance is evaluated using the two solvers and a score value (to be minimised) is calculated. The default setting of irace compares configurations based on ranking. Therefore, the absolute difference between score values is not important. Details of the scoring are as follows.

- If the generator configuration is unsatisfiable, then a special infinite score value is returned. irace will discard the configuration immediately.
- If the generator configuration is too large be to translated (Savile Row is out of time or memory), or not solvable by `minion`, then the score is set to 2.
- If the generator configuration is satisfiable and an instance is found:
  - if the instance is unsatisfiable or too large to be translated (Savile Row is out of time or memory for either solver), then a score of 1 is returned,
  - if the instance is too difficult for the favoured solver, or too easy for the base solver, then a score of 0 is returned, or

- if the instance is solved within the given time and memory limits, then the negation of the ratio between the solving time of the base and the favoured solvers is returned.

4.3 Experimental setup

The memory limit given to each evaluation is 7GB. The time limit for Savile Row and `minion` is 5 minutes each. The time limit for the favoured solver is 5 minutes, while the base solver is allowed between 10 seconds and 25 minutes. Chuffed version 0.10.3 and CPLEX version 12.9 are used. Instances are translated to Chuffed directly via Savile Row. CPLEX input is translated to MiniZinc format first using Savile Row, and then to CPLEX input format using MiniZinc [6]. The compilation time required by MiniZinc was never more than a few seconds, as the input MiniZinc files have been pre-processed and optimised by Savile Row.

Solving time on an instance is calculated as the average value across three runs. Each experiment is run on a cluster node with two 2.1 GHz Intel Xeon E5-2695 processors. Since irace supports parallelism, 30 cores are used per experiment. Each tuning is given a budget of 5000 evaluations and 48 hours of wall-time, and is stopped when either of the two budgets is exhausted.

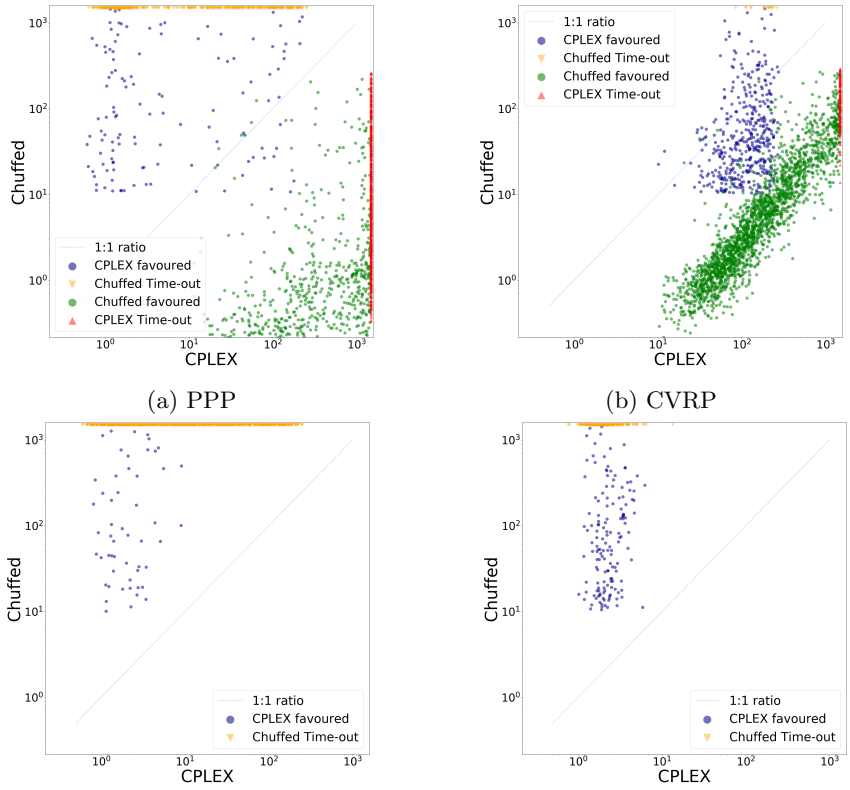
5 Results

The discriminating instances found[2] for each problem class are plotted in Figure 1. Table 1 details how many evaluations the tuning spent on each type of instance, with numbers describing what the search space looks like during tuning.

For CVRP and PPP, we found discriminating instances for both CPLEX and Chuffed. However, the number of instances found in the Chuffed-favoured experiment is larger than in the CPLEX-favoured experiment (≈ 2000 vs. ≈ 50 instances for CVRP, and ≈ 1400 vs. ≈ 600 instances for PPP). Detailed results on the search space during the tuning show that in the Chuffed-favoured experiment, the majority of evaluations is spent on instances solved by Chuffed within 300 seconds, while in the CPLEX-favoured experiment, the majority is spent on instances where CPLEX timed out (both problem classes), or on instances that are very easy for Chuffed (CVRP only). This indicates that in our current setting, although instances where CPLEX is better than Chuffed on those two problem classes exist (and are found by our tuning), overall Chuffed is better at solving these problems than CPLEX.

For the two remaining problems, Transshipment and Warehouse Location, we see a different picture. Many CPLEX-favoured instances are found, while there are no Chuffed-favoured instances at all. For Transshipment, in the majority of evaluations in both tuning experiments either Chuffed times out or the generator configuration was not solved in time. Figure 1 also shows that CPLEX's solving time on the discriminating instances of the same problem is quite small (≤ 10

[2] Code and data are at: https://github.com/stacs-cp/CPAIOR2020-InstanceGen



(a) PPP (b) CVRP

(c) Transshipment (d) Warehouse Location

Fig. 1: Solving time of Chuffed and CPLEX on the discriminating instances found. We run irace twice with each solver favoured in turn. The plotted time-outs are only for the base solver.

seconds). These observations suggest that CPLEX completely dominates Chuffed on Transshipment, which is exactly what we expected. For Warehouse Location, the story is a bit different. Instances in the CPLEX-favoured experiment are mostly too easy for Chuffed. A more detailed look at those instances reveals that Chuffed is comparable to CPLEX on the instances that are "too easy". Therefore, we conjecture that CPLEX may not completely dominate Chuffed on Warehouse Location, but it does dominate Chuffed on the "more difficult" instances which take at least 10 seconds to solve by Chuffed in our setup.

6 Feature analysis

To gain more insights into the discriminating instances found, we extract their FlatZinc features using the `fzn2feat` tool (part of `mzn2feat`[5]). There are 95

	gen	SR-timeout	base	favoured	within the range			
	failed	& unsat	easy	timeout	ratio≤1	ratio>1	ratio>2	ratio>10
PPP-chuffed	85	932	538	554	4	1474	1471	1419
PPP-cplex	67	1019	301	1290	14	670	665	597
CVRP-chuffed	457	178	1117	741	4	2225	2224	2080
CVRP-cplex	686	178	1768	1865	330	110	59	8
TP-chuffed	1762	30	323	2862	0	0	0	0
TP-cplex	2108	479	50	643	0	1096	1096	1038
WLP-chuffed	626	58	4277	34	0	0	0	0
WPL-cplex	819	298	1174	0	0	519	518	481

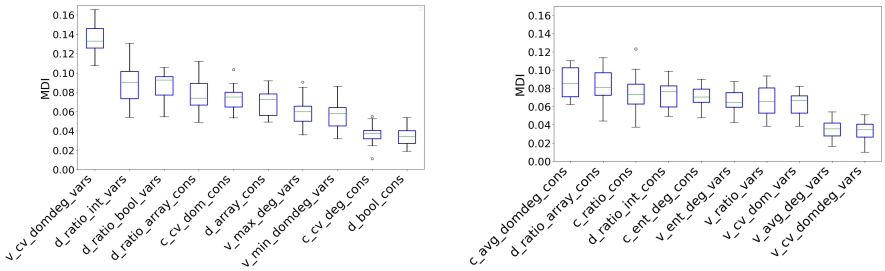
Table 1: Number of runs for each instance type during tuning. Experiment name has the problem class and the favoured solver. Columns: `gen failed` (unsolved generator configurations), `SR-timeout&unsat` (Savile Row timed out or the instances are unsatisfiable), `base easy` (solved by the base solver within 10s), `favoured timeout` (the favoured solver timed out), `within the range` (sat instances solved within 300s by the favoured solver and not solved within 10s by the base solver). The four final numbers for each experiment show how discriminating instances are; `ratio` is solving time of the base to the favoured solver. If the base solver times out, 25min is used.

features grouped into 6 categories (variables, constraints, domains, global constraints, objective, and solving features) [5]. For each of the two problem classes where discriminating instances are found, CVRP and PPP, we use the Balanced Random Forest classifier from the Python package `imblearn` [21] with 200 estimators and 5-fold cross validation. To identify the most important features representing the discriminating property between the two solvers, Mean Decrease Impurity [23] of each feature is calculated across 20 runs. Random Forests have been shown to be the overall best choice for modelling running time of CP and SAT solvers [16]. The Mean Decrease Impurity (MDI) is a widely-used measurement for feature importance analyses in Random Forest models. The MDI of a feature in each tree is the weighted decrease in impurity (using Gini importance) across all tree nodes where the feature is used in the splits. The overall MDI of a feature is calculated by averaging the MDI values across all trees in the forest.

To avoid noise in the measurement of solving time, we only consider discriminating instances where the ratio between the solving time of the bad solver and the good solver is larger than 1.5. Each instance is labelled as either Chuffed-favoured or CPLEX-favoured.

Figure 2 show the importance values of the top 10 features for PPP and CVRP. For PPP, the first feature, `v_cv_domdeg_vars`, shows a much higher importance value compared to the rest. This feature defines the Coefficient of Variance of the ratios between domain size and degree (number of constraints involved) of all variables. A more detailed look into the data indicates that Chuffed-favoured instances tend to have similar ratios between domain size over degree across different variables, while for CPLEX-favoured instances those ratios differ more drastically between variables. For CVRP, there is no clearly distinguished single feature. Moreover, the list of the most important features varies between

the two different problem classes. This suggests that the favouring-behaviours of the two solvers depend on the problem.



(a) PPP (b) CVRP

Fig. 2: Top 10 FlatZinc features for discriminating instances based on MDI

7 Conclusion and future work

In this work, we presented an automated instance generation system that can produce discriminating instances between two solvers. We demonstrated our method on four problem classes with the CP solver Chuffed and the MIP solver CPLEX. This revealed the strengths and weaknesses of each solver. A further analysis of the discriminating instances using FlatZinc features [5] suggests that the discriminating behaviour is problem dependent.

In future work, we plan to extend our system for finding discriminating instances for a portfolio of more than two solvers; the open question here is how to define the discriminating property. We also plan more detailed feature analysis investigating the relationship between solver performance and instance space. This can involve defining new instance features based on the high-level types supported by Essence, which may provide more high-level structural information about the instances. Finally, we currently rely on irace's built-in exploration of the generator configuration space to ensure the diversity of the generated instances. This can be improved by investigating more advanced approaches for controlling instance diversity more directly, including incorporating a diversity measurement such as multi-objective indicators [25] into the scoring values of the tuning, or forcing each generator configuration to generate instances far away from the current instance set by adding constraints into the generator model.

Acknowledgements This work is supported by EPSRC grant EP/P015638/1 and used 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).

References

1. Akgün, Ö., Dang, N., Miguel, I., Salamon, A.Z., Stone, C.: Instance generation via generator instances. In: Schiex, T., de Givry, S. (eds.) CP 2019: Proceedings of the 25th International Conference on Principles and Practice of Constraint Programming. LNCS, vol. 11802, pp. 3–19. Springer (2019). https://doi.org/10.1007/978-3-030-30048-7_1
2. Akgün, Ö., Frisch, A.M., Gent, I.P., Hussain, B.S., Jefferson, C., Kotthoff, L., Miguel, I., Nightingale, P.: Automated symmetry breaking and model selection in Conjure. In: CP 2013: Proceedings of the 19th International Conference on Principles and Practice of Constraint Programming. LNCS, vol. 8124. springer (2013). https://doi.org/10.1007/978-3-642-40627-0_11
3. Akgün, Ö., Miguel, I., Jefferson, C., Frisch, A.M., Hnich, B.: Extensible automated constraint modelling. In: AAAI 2011: Proceedings of the Twenty-Fifth AAAI Conference on Artificial Intelligence. pp. 4–11. AAAI Press (2011), https://www.aaai.org/ocs/index.php/AAAI/AAAI11/paper/viewPaper/3687
4. Alissa, M., Sim, K., Hart, E.: Algorithm selection using deep learning without feature extraction. In: GECCO 2019: Proceedings of the Genetic and Evolutionary Computation Conference. pp. 198–206. ACM (2019)
5. Amadini, R., Gabbrielli, M., Mauro, J.: An enhanced features extractor for a portfolio of constraint solvers. In: SAC 2014: Proceedings of the 29th Annual ACM Symposium on Applied Computing. pp. 1357–1359. ACM (2014). https://doi.org/10.1145/2554850.2555114, code available from https://github.com/CP-Unibo/mzn2feat
6. Belov, G., Stuckey, P.J., Tack, G., Wallace, M.: Improved linearization of constraint programming models. In: Rueher, M. (ed.) CP 2016: Proceedings of the 22nd International Conference on Principles and Practice of Constraint Programming. LNCS, vol. 9892, pp. 49–65. Springer (2016). https://doi.org/10.1007/978-3-319-44953-1_4
7. Chu, G., de la Banda, M., Stuckey, P.: Exploiting subproblem dominance in constraint programming. Constraints **17**(1), 1–38 (2012). https://doi.org/10.1007/s10601-011-9112-9, code available from https://github.com/chuffed/chuffed
8. Dantzig, G.B., Ramser, J.H.: The truck dispatching problem. Management Science **6**(1), 80–91 (1959). https://doi.org/10.1287/mnsc.6.1.80
9. Drexl, A., Nissen, R., Patterson, J.H., Salewski, F.: 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), 59–72 (2000)
10. Frisch, A.M., Harvey, W., Jefferson, C., Martínez-Hernández, B., Miguel, I.: Essence: A constraint language for specifying combinatorial problems. Constraints **13**(3), 268–306 (2008). https://doi.org/10.1007/s10601-008-9047-y
11. Gent, I.P., Jefferson, C., Miguel, I.: Minion: A fast scalable constraint solver. In: Proceedings of ECAI 2006. pp. 98–102. IOS Press (2006), http://ebooks.iospress.nl/volumearticle/2658
12. Gent, I.P., Walsh, T.: CSPlib: A benchmark library for constraints. In: Jaffar, J. (ed.) CP 1999: Proceedings of the 5th International Conference on Principles and Practice of Constraint Programming. LNCS, vol. 1713, pp. 480–481. Springer (1999). https://doi.org/10.1007/978-3-540-48085-3_36, available from http://www.csplib.org

13. Hooker, J.N.: Testing heuristics: We have it all wrong. Journal of heuristics **1**(1), 33–42 (1995)
14. Hoos, H.H., Kaufmann, B., Schaub, T., Schneider, M.: Robust benchmark set selection for boolean constraint solvers. In: International Conference on Learning and Intelligent Optimization. pp. 138–152. Springer (2013)
15. Horie, S., Watanabe, O.: Hard instance generation for sat. In: International Symposium on Algorithms and Computation. pp. 22–31. Springer (1997)
16. Hutter, F., Xu, L., Hoos, H.H., Leyton-Brown, K.: Algorithm runtime prediction: Methods & evaluation. Artificial Intelligence **206**, 79–111 (2014)
17. IBM: IBM ILOG CPLEX Optimization Studio documentation (2019)
18. Julstrom, B.A.: Evolving heuristically difficult instances of combinatorial problems. In: GECCO 2009: Proceedings of the 11th Annual conference on Genetic and Evolutionary Computation. pp. 279–286. ACM (2009). https://doi.org/10.1145/1569901.1569941
19. Kotthoff, L.: Algorithm selection for combinatorial search problems: A survey. In: Data Mining and Constraint Programming, pp. 149–190. Springer (2016)
20. Laporte, G.: The vehicle routing problem: An overview of exact and approximate algorithms. European Journal of Operational Research **59**(3), 345–358 (1992). https://doi.org/10.1016/0377-2217(92)90192-C
21. Lemaître, G., Nogueira, F., Aridas, C.K.: Imbalanced-learn: A python toolbox to tackle the curse of imbalanced datasets in machine learning. Journal of Machine Learning Research **18**(17), 1–5 (2017), http://jmlr.org/papers/v18/16-365.html
22. López-Ibáñez, M., Dubois-Lacoste, J., Cáceres, L.P., Birattari, M., Stützle, T.: The irace package: Iterated racing for automatic algorithm configuration. Operations Research Perspectives **3**, 43–58 (2016). https://doi.org/10.1016/j.orp.2016.09.002, http://iridia.ulb.ac.be/irace/
23. Louppe, G., Wehenkel, L., Sutera, A., Geurts, P.: Understanding variable importances in forests of randomized trees. In: Advances in neural information processing systems. pp. 431–439 (2013)
24. Nethercote, N., Stuckey, P.J., Becket, R., Brand, S., Duck, G.J., Tack, G.: MiniZinc: Towards a standard CP modelling language. In: CP 2007: Proceedings of the 13th International Conference on Principles and Practice of Constraint Programming. LNCS, vol. 4741, pp. 529–543. Springer (2007). https://doi.org/10.1007/978-3-540-74970-7_38
25. Neumann, A., Gao, W., Wagner, M., Neumann, F.: Evolutionary diversity optimization using multi-objective indicators. In: GECCO 2019: Proceedings of the Genetic and Evolutionary Computation Conference. pp. 837–845. ACM (2019)
26. Nightingale, P., Akgün, Ö., Gent, I.P., Jefferson, C., Miguel, I., Spracklen, P.: Automatically improving constraint models in Savile Row. Artificial Intelligence **251**, 35–61 (2017). https://doi.org/10.1016/j.artint.2017.07.001
27. Orden, A.: The transhipment problem. Management Science **2**(3), 276–285 (1956). https://doi.org/10.1287/mnsc.2.3.276
28. Smith-Miles, K., Baatar, D., Wreford, B., Lewis, R.: Towards objective measures of algorithm performance across instance space. Computers & Operations Research **45**, 12–24 (2014)
29. Smith-Miles, K., van Hemert, J.: Discovering the suitability of optimisation algorithms by learning from evolved instances. Annals of Mathematics and Artificial Intelligence **61**(2), 87–104 (2011). https://doi.org/10.1007/s10472-011-9230-5
30. Smith-Miles, K., Lopes, L.: Generalising algorithm performance in instance space: A timetabling case study. In: International Conference on Learning and Intelligent Optimization. pp. 524–538. Springer (2011)

31. Smith-Miles, K., Lopes, L.: Measuring instance difficulty for combinatorial optimization problems. Computers & Operations Research **39**(5), 875–889 (2012)
32. Vanhoucke, M., Maenhout, B.: NSPLiba nurse scheduling problem library: a tool to evaluate (meta-) heuristic procedures. In: Operational research for health policy: making better decisions, proceedings of the 31st annual meeting of the working group on operations research applied to health services. pp. 151–165 (2007)