Submitted to *Operations Research*
manuscript OPRE-2010-07-324.R5

Generating Applicable Synthetic Instances for Branch Problems

Leo Lopes
SAS Institute, USA, leo.lopes@sas.com

Kate Smith-Miles
School of Mathematical Sciences, Monash University, Australia, kate.smith-miles@monash.edu

Generating valid synthetic instances for branch problems – those that contain a core problem like Knapsack or Graph Coloring, but add several complications – is hard. It is even harder to generate instances that are applicable to the specific goals of an experiment and help to support the claims made. This paper presents a methodology for tuning instance generators of branch problems so that synthetic instances are similar to real ones and are capable of eliciting different behaviors from solvers. A statistic is proposed to summarize the applicability of instances for drawing a valid conclusion. The methodology is demonstrated on the Udine Timetabling problem. Examples and the necessary cyberinfrastructure are available as a project from Coin-OR.

Key words: Instance generation; Data mining; Optimization; Timetabling

1. Introduction

In OR research, claims are often challenged based on issues regarding the instances used, but these objections are seldom expressed precisely enough to be actionable by authors. In OR practice, algorithms are often deployed on real instances when they have only been tested on benchmark synthetic instances. The research reported on here accomplishes two main goals: it helps create instances for computational studies that are more applicable to the claims made; and it helps reduce the risk of deployment failure for practical algorithms. The term *applicability* of a dataset is defined in Hall and Posner (2010) as a principle of effective instance generation: "The generation scheme generates the types of data sets that are needed".

The main step is a test that establishes the degree to which a set of instances satisfies requirements relevant to the claims of the study:

1) that synthetic instances are similar to real instances, where claims are made about the suitability of an algorithm for real-world deployment;

2) that instances be discriminating of solver performance (i.e., that the studied solvers perform differently on the same instance), where claims are made about the strengths and weaknesses of solvers; and

3) that this discriminating power cannot be easily traced to known results, where claims are made about new explanations of solver performance.

These goals promote computational results that are relevant, defensible, and novel. As an example, consider two timetabling algorithms which differ in the way they solve the vertex coloring subproblems that arise as a result of conflicts in instructor availability. Algorithm A uses a heuristic while Algorithm B uses a combination of integer and constraint programming. If the study aims to demonstrate which algorithm is better suited to realistic timetabling instances, then any synthetic instances used must be demonstrably similar to real-world instances. If the study intends to explain the strengths and weaknesses of each algorithm for classes of instances, then the instances will need to be discriminating of solver performance, while ensuring that any discriminatory power is not only due to known results. It is not difficult to exploit known results to construct a set of test instances where one algorithm is superior to another, but this evaluation will not support novel claims, and researchers should take steps to ensure they can demonstrate that their choice of test instances is not biased. Suppose the performance of Algorithm A is significantly better than Algorithm B's when the vertex coloring instances are close to being bipartite. This is potentially uninteresting if Algorithm A is based on the degree saturation heuristic DSATUR, since it is a known result that DSATUR is exact for bipartite graphs (Brélaz 1979). If the only instances used to test the performance of the two algorithms were bipartite graphs, then the conclusion that Algorithm A is superior should be challenged due to the biased nature of the test instances. Using a broader instance set containing non-bipartite graphs would be more applicable to a novel study of the effectivess of these two algorithms. If the difference in algorithm performance is not explained using all we know in the literature, then the instance set can support new insights where they can be found, and the instance set can be considered unbiased.

The most broadly applicable instance sets meet all three requirements. Depending on the claims made within an individual study, a subset of the requirements may be satisfactory. The approach taken here is to utilize data mining techniques to establish the degree to which a set of synthetic instances meets some or all of these requirements. The test works by collecting relevant statistical features related to known

results from a set of real instances and a set of synthetically generated instances, and examines how readily these instances can be separated or partitioned in the feature space. When it is easy to separate synthetic instances from real ones, or when solver discrimination can be traced to a known feature, then a precise way to communicate the shortcomings of the instances in terms of its features is offered. This feedback goes beyond demonstrating that an instance set has shortcomings, and offers constructive and specific advice on how to improve applicability of instances to support the claims.

The exposition of computational results has well-documented shortcomings (Hall and Posner 2010, Hooker 1995, Johnson 2002, Moret 2002, Anderson 2002). When instances are not chosen carefully, the conclusions of the study may be challenged. Tuning solvers to benchmark instances can result in poor performance outside that set, especially when the benchmark instances are not particularly diverse across a spectrum of difficulty (Hill and Reilly 2000). Understanding the boundaries of algorithm performance, and reporting those boundaries when claiming superiority of an algorithm, is critical for good practice of computational reporting. Recent developments in measuring the size of an algorithm's footprint (Corne and Reynolds 2011, Smith-Miles and Tan 2012), defined as the region in instance feature space where the algorithm's observable good performance can be convincingly generalized, provide important tools for measuring the extent to which a claim of superior performance can be supported. Effective methods for generating instances that reveal the strengths and weaknesses of algorithms are critical to enable such algorithm footprints to be established. To avoid deployment disasters it is also important to establish if the footprint of an algorithm includes the set of real-world instances.

The focus of this research is on generating instances for *branch* problems. The terms *branch, core*, and *leaf* problems have been defined (Lopes 2010) to describe classes of problems based on how abstract or concrete they are. In brief, a branch problem is one step removed from a specific problem tied to a specific customer, which would be a *leaf* problem. Like leaf problems, branch problems contain many conflicting goals. A branch problem usually combines one or more core problems, some of which may be hard, like Knapsack, Graph Coloring, or Bin Packing, and some easy, like Sorting, or Shortest Paths, with other constraints and objectives.

Libraries of benchmark problems (e.g., Beasley (1990), Ariyawansa and Felt (2004)) are often used to test solvers for branch problems. Using libraries leads to another set of problems, like over-tuning solvers to

a relatively small set of instances and the aging of the instances. Branch problems are almost always $\mathcal{NP}$-hard,and instance generation is even more challenging due to their complicated multinomial distributions. Even valid instances are sometimes hard to generate because the data that define each instance are the result of complex (and sometimes peculiar) processes, resulting in joint distributions which are hard to approximate using standard distributions. Unlike in core problems, in branch problems a single statistic is typically insufficient to describe phase transitions (Monasson et al. 1999, Achlioptas et al. 2005) between easy and hard instances. It is in this complicated environment that we provide a methodology for generating instances of branch problems that are applicable to the claims made.

Most of the existing work on applicable instance generation focuses on core problems. Hooker (1995) suggested that it is not possible to generate synthetic instances that are realistic. Instead, the author emphasized writing generators that can reliably reproduce a specific feature of an instance, so that the impact of that feature on algorithm performance can be measured experimentally. Reilly (2009) pointed out the importance of correlation for characterizing the hardness of instances and presented a procedure that can generate diverse instances in which parameters have a given correlation, for problems like Knapsack, assignment, set covering, and others. Its focus is on creating instances that are challenging for solvers. Smith-Miles and van Hemert (2011) is subtly different: the focus is on creating instances via evolutionary algorithms that can discriminate the performance of two solvers. However, these studies do not address whether or not the instances generated are likely to appear in practice.

Hall and Posner (2001) emphasize that when the goal is to predict real-world performance, then the instances generated should take into account the complex distributions that arise from real business processes. Leyton-Brown et al. (2009) generate instances of a branch problem from simulations of realistic processes, and use importance sampling to tune the parameters of the generator so that the instances become hard. Taillard (1995) also goes beyond sampling from simple distributions, and uses an iterative algorithm during the simulation process, to generate what the author calls "real-world like" quadratic assignment problem (QAP) instances. Those instances were generated so that the value of a feature of the instances matches the predicted value of that feature for realistic instances of the same size.

Adding to the challange of generating applicable instances for branch problems, we highlight also the need to develop a metric to demonstrate the success of the instance generation process. The research

presented in this paper associates a concrete, measurable definition with the principle of applicability in the context of branch problems: an instance set is most applicable to an experiment if it is realistic and differences in solver performance cannot be traced to a feature that is already known. Realism is defined in terms of statistical features of the instances. If the set of real instances and the set of synthetic instances cannot be separated (partitioned in feature space) using all that is known about the problem, then the synthetic instances are deemed realistic.

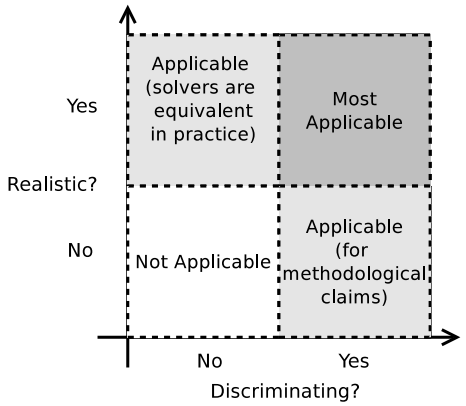
The remainder of this paper is organized as follows. In Section 2 we provide a generic description of the proposed approach. This is followed by an example application. In Section 3 we discuss the Udine Timetabling problem, its relationship to Vertex Coloring, an instance generator (Burke et al. 2010b) that is used to generate synthetic instances, an experiment that can pinpoint in great detail what aspects of an instance are not realistic, and how to use the outcome of the experiment to generate a new set of instances that addresses the specific shortcomings identified. In Section 4 we present a statistic that can be used, with care, to summarize the applicability of an instance set. Section 5 offers some insights into how to strengthen a study by using these techniques, and discusses some questions that remain open. An electronic compendium to this paper provides details of an open source package, available from Coin-OR (COIN-OR 2000), that implements the procedures proposed here.

2. Defining applicability

This paper address the following problem: Is a set of synthetic instances S applicable to a claim that the performance of one algorithm or model is superior to that of another one on instances like those in the given set R? If not, then what features separate S from instances that are applicable, and how can we modify the instance generator to produce more applicable instances? Ideally, we would also like to ensure that any explanations of this superior performance are novel and interesting, and not just corollaries of known results.

An instance that contains known features that have been exploited by one algorithm and not others will produce a biased comparison. An instance whose features are substantially different from those observed in real instances will produce an inaccurate comparison. Applicability as defined in this paper protects against these flaws simultaneously. The methodology can also cover the easier special cases when only one flaw needs to be addressed.

Figure 1 The horizontal axis measures the discriminating power of S. The vertical axis measures the realism of S. The closer an instance set S is to the upper-right corner, the more applicable it is to strong conclusions.



Several highly cited papers (Johnson 2002, Bartz-Beielstein et al. 2004, Hooker 1995, Hall and Posner 2007) provide comprehensive conceptual advice on designing computational experiments. In these papers, different authors use different terms for similar concepts. An annotated bibliography on algorithm experimentation can be found in McGeoch and Amherst (2002). We choose to use the definitions of applicability in Hall and Posner (2010) when possible for several reasons: it is the latest contribution from authors who have developed related ideas for over a decade (Hall and Posner 2001, 2007); it provides a more specific focus on instance generation than some other papers; it contains a comprehensive review of existing work in this area; and it is part of an entire book dedicated to issues related to empirical testing of optimization algorithms (Bartz-Beielstein et al. 2010). The definition of applicability used in this paper depends on two dimensions: whether an instance set can discriminate solver performance as relevant to the experiment; and whether it is realistic. An instance set is said to be discriminating if the performance of the solvers in the study is significantly different on instances from that set; otherwise it is said to be non-discriminating. We learn little about the strengths and weaknesses of optimization solvers by studying non-discriminating instances. An instance set may be discriminating along one performance dimension (e.g. time to find first feasible solution) but not along another (time to prove optimality). An instance is realistic if it is "similar" in feature space to instances known to be generated from a business process.

Figure 1 illustrates the definition of applicability. In Hall and Posner (2010), applicability is further refined into the three generation principles described in Table 1: Completeness; Parsimony; and Comparability. In this framework, completeness depends on the diversity of R. As long as R contains sufficient

Table 1 The three principles of instance generation from Hall and Posner (2010) and how they influenced this research.

Principle	Definition	Implementation
Completeness:	All important instances can be generated	Generate a set S that generalizes the known-real set R and allows strong statistical conclusions.
Parsimony:	The variations in the instances are those that are important for the experiment	Focus on generating instances that discriminate solvers, and avoid generating instances that elicit tied behaviors in greater proportions than those naturally found in the known-real set.
Comparability:	The experiments are comparable within and between studies	Ensure replicability by: defining a statistic for the overall quality of the instance set that provides researchers with at least a coarse notion of instance set applicability; publish the generator as open source; report random seeds used to generate the instance sets.

diversity, then S will contain at least that diversity. Parsimony is addressed in part by using the concept of discrimination (rather than raw performance) and in part concurrently with bias below (this differs from the principles in Hall and Posner (2010)); and comparability is addressed by creating a summary statistic that measures the applicability of an instance set in terms of realism and discriminating ability. An additional way to ensure comparability is to publish the generators themselves as open source.

The purpose of instance generation is to create instances that help draw conclusions from an experiment (see parsimony in Table 1, and Hooker (1995)). The No-Free-Lunch Theorem (Wolpert and Macready 1997) postulates that two algorithms can only perform differently if they use different prior knowledge about the distribution of instances. While the necessary conditions for the theorem rarely hold for branch problems (Koehler 2007), one of its insights probably does: interesting new algorithms outperform existing ones because they exploit some previously unknown feature that is sufficiently important and appears sufficiently often. Thus, performance discrimination is an essential feature of an applicable instance set for methodological claims.

Bias is widely discussed in the literature. The main concern is that synthetic instances may systematically favor one solver over another. To avoid this form of bias, this research does not consider which solver won, only whether their performance was different. It is the main experiment of a paper, separate from the one that establishes whether instances are of sufficient quality, that should address benefits of one solver over another and explain them.

An instance set may suffer from yet another type of bias: if it contains only discriminating instances, then the differences between the solvers may be exaggerated in the study's conclusions. If the goal is to

determine which solver is better and how frequently, some non-discriminating instances should be retained, roughly in the same proportion as they are found in the real set. If the goal of the experiment is simply to explain the differences in solver performance across instance space, then this restriction is not needed.

For the broad problem stated at the beginning of Section 2, it is not sufficient to demonstrate in separate steps that synthetic instances cannot be separated from real ones and that discriminating instances cannot be separated from non-discriminating ones using a set of features that captures all known results. Even when separation fails along discrimination and realism dimensions considered one at a time, it may still be the case that some discriminating instances are not realistic. If the two dimensions are considered concurrently for the same instance, then the results are stronger. Thus, in this research the techniques chosen use three classes: realistic (R), whether discriminating or not; synthetic discriminating (SD); and synthetic non-discriminating (SN). An instance set upon which separation fails in this mode results in the strongest possible claims regarding its applicability for comparing the algorithms and establishing their suitability for real world implementation.

Discrimination (whether or not one solver predominantly wins) is a more precise and useful concept than hardness for determining applicability. Using discrimination enables greater parsimony: instances not important for the experiment, such as those that are too hard for any solver to address and those on which any reasonable solver would behave essentially the same way can be discarded if they are not relevant to the experiment.

In summary:

- If S is not discriminating and not realistic, then it is not applicable to any experiment claiming to show either a methodological contribution or practical relevance. If S is discriminating and realistic, then it is most applicable. If S falls within the other two quadrants, S may still be applicable, depending on the claim. If S is realistic, but not able to discriminate solvers, then the study can still have a strong conclusion: the solvers are practically equivalent on problems like those in R. If S is discriminating, but not realistic, then as long as the claims of the study are purely methodological a strong conclusion can still be supported by the instances.

- The synthetic set used to test the algorithms should contain both discriminating and non-discriminating instances, and it should be hard to separate instances into discriminating, non-discriminating, and real based on all known problem features. Using all known features prevents claiming

a new insight defining when solver A is better than B, or vice-versa, when the reason for the difference in performance was already known and could be traced easily to instances that had a certain characteristic. When analyzing results, it may be useful to know which features distinguish discriminating from non-discriminating instances (e.g., for performance prediction), but that is, and should be, a separate step from evaluating the applicability of the instance set. Of course, if we wished to find properties of the features (existing or novel) that distinguish discriminating from non-discriminating instances, or that distinguish instances where one solver outperforms another solver, then we do not need to insist on generating instances that can't be separated in feature space by performance labels. In that case, our goal would be to find the rules defining such separations based on features, as we have done in earlier work (Smith-Miles and van Hemert 2011), where we have intentionally evolved instances that can be easily discriminated in feature space based on algorithm performance. If no such separation is possible using well studied features, but separation can be achieved using a novel feature set, then we have contributed new insights into the features that distinguish algorithm performance. Demonstrating separation on existing well-studied features is less likely to make a new contribution and is less applicable to new claims. Insisting that the existing literature-based feature set (Smith-Miles and Lopes 2012) cannot separate discriminating and non-discriminating instances though, ensures that we do not simply rediscovered known results, and researchers have a set of instances applicable to claims of new insights.

2.1. Using Classification Trees to separate instances

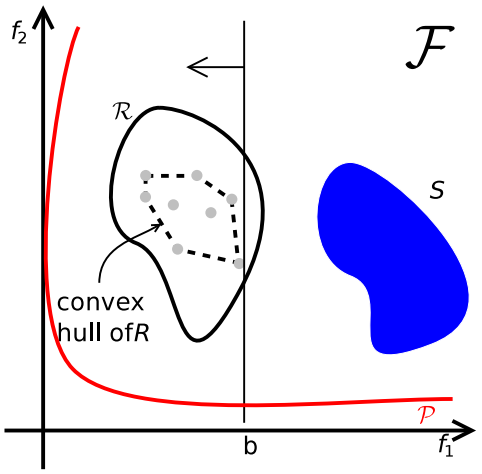
To establish applicability for the broad problem posed at the start of Section 2, we must answer yes to three questions:

1. Does S contain discriminating instances?
2. Does S contain realistic instances?
3. Is the discrimination power of S not easily traceable to a well-known result?

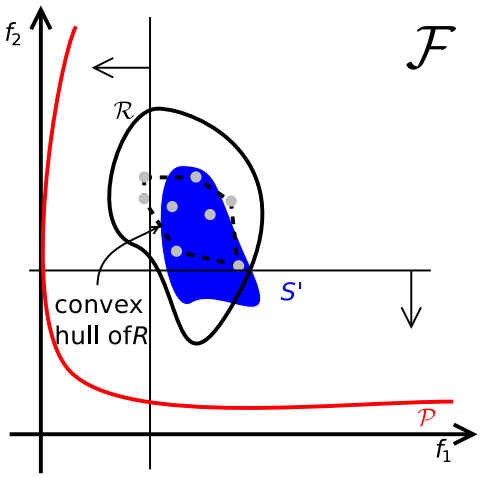
Furthermore, if the answer to any of these questions is no, we must offer constructive guidance on how to improve the instance set. This research uses classification trees (Quinlan 2003) to address the realism and discriminating power of S, primarily because: 1) it provides guidance that can be incorporated into instance generators; 2) it automates the two natural procedures for examining multi-factorial data (i.e.,

Figure 2 A plot of problems (instance sets) $\mathcal{P}$ and $\mathcal{R}$, real instances R (in this case with eight instances), and synthetic instance sets S and S' plotted on a 2-dimensional feature space $\mathcal{F}$. Instance set S in Figure 2a is not realistic. Instance set S' in Figure 2b is more realistic than S, although it still covers only parts of $\mathcal{R}$ and contains instances not in $\mathcal{R}$. All instances are in the generic problem set $\mathcal{P}$.

(a) The hyperplane $f_1 \leq b$ indicates that the feature on the horizontal dimension makes S easy to separate from R



(b) These are the best orthogonal hyperplanes at separating R (the vertical hyperplane) and S' (the horizontal hyperplane).



logistic ANOVA followed by separation – although typical implementations use information-theoretical approaches to accomplish essentially the same outcome); 3) it is robust to effects of factor combinations; 4) it is computationally tractable; and 5) it is easily available in many statistical packages including R (R Development Core Team 2010) and SAS.

The framework below, illustrated in Figure 2, describes how the procedure works. $\mathcal{R}$ is the set of all real instances of the problem, whereas $\mathcal{P}$ is the set of all instances of the problem, whether real or not; $R \subset \mathcal{R}$ is a set of instances known to be real. A synthetic set of instances $S \subset \mathcal{P}$ is realistic if it cannot easily be separated from R using the characteristics of $\mathcal{P}$ and $\mathcal{R}$ known to be important. Much is already known and documented about what distinguishes instances of particular optimization problems (Smith-Miles and Lopes 2012) and the features that can reveal differences between real and synthetic instances (Tuson and Harrison 2005).

To use the framework, the experiment must provide:

- A seed set $R \subset \mathcal{R} \subset \mathcal{P}$.

The set $\mathcal{R}$ usually can *not* be described by a concise formulation, although current state of the art often assumes that it can. Instead, it is a larger problem $\mathcal{P}$ that can be described by that formulation. $\mathcal{P}$ includes all valid instances, whether applicable or not. For example, all instances of the Traveling Salesman Problem (TSP) can be described as $\mathcal{P} = \{(V, c : V \times V \to \mathbb{Z}, L \in \mathbb{Z})\}$, where V is a set of nodes, c is a cost function associating each edge of the complete graph to an integer, and L is a number. In this decision problem version, the TSP is said to have a solution 'yes' if there exists a tour on V with length less than L. This description of the TSP contains subsets that are clearly not applicable to any claim, such as those where $c(u, v) = 1 \, \forall u, v \in V$. It also contains other subsets that are not applicable in more subtle ways (e.g., delivery problems and chip design problems are both TSPs (in $\mathcal{P}$), but can be different in important ways). The ways in which particular instances become not applicable are dependent on the claims drawn from the experimental results. There currently is no analytical way to describe only the interesting subsets of a problem. Thus the need to provide the procedure with an explicit subset R (for real) instances, which should be as large and diverse as possible.

- A set of statistical features $\mathcal{F}$ known to be relevant to characterizing $\mathcal{R}$.

It may be possible to easily characterize performance of core problems using a couple of statistics from the instance (Smith-Miles and Lopes 2012). In branch problems, there are many different characteristics of an instance that may drive performance. Thus, this research presumes a well-developed literature on $\mathcal{P}$, or at the very least a well-developed literature on a family $\{\mathcal{P}_1, \mathcal{P}_2, \ldots, \mathcal{P}_n\}$ of problems related to $\mathcal{P}$ either in a computational complexity sense or in an application sense. For example, ideally we would have a well developed literature on Udine Timetabling for the example in Section 3. In its absence, literature on graph coloring and on other timetabling variants is used. The procedure also assumes that R, a seed set of real instances drawn from $\mathcal{R}$, is available.

- A generator g that can create a set S of synthetic instances.

The goal is to produce a generator g that is as good as possible. This practical goal drives the choice of a classification tree as the data mining technique.

The proposed algorithm analyzes the instance data from real-world instances and an initial synthetic generator, and iteratively modifies the generator based on feedback from the classifier. The feedback changes the properties of identified features that cause the real-world instances (R) to be easily separated

from the synthetic instances (S) – both discriminating (SD) and non-discriminating (SN) – so that the synthetic instances become more real-world-like. Additionally, the classification tree identifies properties of features that cause the discriminating instances to be easily separated from the non-discriminating instances. If no such properties of the feature set can be found to distinguish discriminating from non-discriminating instances, then we can conclude that there is no existing feature that distinguishes algorithm performance based on known results, and the instance set has the potential for new insights to be created.

Classification trees (Quinlan 2003) split $\mathcal{F}$ using intersections of orthogonal half-spaces that maximize purity within a class (see Figure 2b). Certainly, many other statistical approaches can be used to identify if a set of synthetic instances is separable in feature space from a set of real instances, and they may even produce better separations. However, just informing a researcher that their instance set is not applicable, without providing any feedback as to how to improve the generator seems of much less value. For example, a Linear Support Vector Machine (SVM) (Shawe-Taylor and Cristianini 2000) or even a nonlinear SVM is likely to produce even better separation, but the insights would be even harder to read. Generic classifiers with high accuracy on the available instances may also increase the risk of over-fitting. Other advantages of classification trees are that they are fast to compute and non-parametric (Gehrke 2003). Unsupervised learning methods, such as self-organizing maps (Kohonen 2002), can be used for clustering instances in a high-dimensional feature space and showing similarity (based on defined metrics) in a two-dimensional projected plane (Smith-Miles and Lopes 2011), but these methods (which are illustrated in Section 3.3) fail to provide enough quantitative feedback. We will use classification trees in our algorithm to modify the instance generator, but we can always use other classification methods later to verify that the three classes cannot be well separated even by more powerful algorithms.

Algorithm 1 provides an overview of the procedure that will be discussed in detail after we introduce an example and the main tools used.

3. An Example: Udine Timetabling

Timetabling is better described as a category of problems. In practice, there are as many different timetabling problems as there are processes requiring tight assignments of people to tasks. In general, timetabling problems involve finding assignments that avoid conflicts. Specific timetabling problems involve additional constraints such as capacities, scheduling, and other complications. This diversity makes

Algorithm 1 A high level overview of the procedure. The goal is to update S at each step, using information provided by the classification tree, so that S and R can no longer be effectively separated.

```
1: procedure GenerateApplicableInstances(R,g)
2:                                                  ▷ R: A set of real instances
3:                                                  ▷ g: an instance generator
4:                                                  ▷ p: proportion of R that is non-discriminating
5:                                                  ▷ F: A set of statistical features identified in the literature as relevant
6:                                                  ▷ ε: user-defined tolerance
7:     repeat
8:         Generate S from g
9:         Partition S into {SD, SN}
10:        Run a decision tree on classes SD, SN, and R
11:        Let f ≤ b be the best rule found to describe S
12:        if f ≤ b separates well then
13:            Update g to produce more instances where f > b
14:        end if
15:    until f ≤ b provides poor separation
16:    if |p − |SN|/|S|| ≥ ε then
17:        Randomly delete sufficient instances from SN or SD to ensure that |SN|/|S| ≈ p.
18:    end if
19: end procedure
```

generating synthetic instances challenging in ways that are different from generating synthetic instances for Knapsack or TSP. For example, in University timetabling problems, it is necessary to assign courses attended by students to rooms, and these rooms must have enough capacity for those courses. If there are too many large rooms, then the corresponding constraints are never binding; if there are two few rooms compared to the course enrollments, then the synthetic problems may be always infeasible.

In these situations, even generating *valid* instances – those where the combination of enrollments and room sizes are compatible – can be hard. Generating *applicable* instances is even harder, because the relationships between course enrollments and class sizes are the product of complex business processes involving competing departments, course popularity, course dependencies, etc. Computational results on *not applicable* synthetic instances may result in the design or selection of poor solvers.

For this research, we chose to focus on the Udine Timetabling problem, also known as Curriculum-based Course Timetabling problem (heretofore CTT, to match others' convention). This problem was used as track 3 of the 2007 International Timetabling Competition (ITC2007), which enabled us to start with an existing generator, and to use existing solvers.

A detailed description of the CTT can be found in Gaspero et al. (2007). CTT can be summarized as follows:

- The input is:

  — a set of courses, each of which is described by: the number of lectures to be scheduled over a week; the number of different days on which the lectures should be assigned; the instructor assigned to it; a set of times when it cannot be scheduled; and the number of students enrolled in it.

  — a set of rooms with given capacity.

  — a set of curricula, each of which represents one or more students taking a course sequence.

- The output is an assignment of courses to rooms and periods.

- The goal is to assign *all lectures* to *rooms* and *timeslots* over a week so that no room, teacher or student is assigned to two lectures during the same timeslot, and no lecture is assigned to a timeslot during which the corresponding teacher is unavailable. There are also a number of other considerations, treated as soft constraints with different penalties in the objective function:

  — Courses should fit in the rooms assigned to them.

  — Lectures of a course ideally should be spread over at least a course-specific number of days.

  — Lectures for courses in the same curriculum ideally should be scheduled consecutively.

  — All lectures for the same course ideally should be assigned the same room.

There are several mixed integer programming (MIP) formulations of the problem above. Some promising formulations can be found in Burke et al. (2010a). MIP-based algorithms and heuristics currently available in the literature are not competitive with the heuristics below on the applicable instances identified in this paper.

In this example, two very different solvers will be compared: a constraint propagation code combined with Simulated Annealing (SACP) written in Java (Müller 2009); and a Tabu Search over a weighed constraint satisfaction problem (TSCS), written in C++ (Nonobe and Ibaraki 2001). Both of these open source codes were provided to us by the authors for use in this study.

The experimental claim to be validated in this example is: *After 10 minutes of running time, SACP produces objective values at least as good as those from TSCS on instances like those from the ITC2007 competition.*

Evaluating this involves a paired-sample hypothesis test on means. Care should be taken with other important details (Johnson 2002), but what needs to be done is well understood. The generator and solvers

are available and the problem is well-defined. The difficult part is establishing that the data (the instances) are applicable to the claim.

The set $\mathcal{R}$ in this example is the set of all timetabling instances that can be described by the formulation in Burke et al. (2010a). R is the set of 21 instances used in the ITC2007 competition. Initially, g is the generator in Burke et al. (2010b). S is the set of instances generated by running g with parameters suggested in the documentation for the instance generator. S' will be a new set of instances based on modifying g with feedback from the classification tree.

3.1. An experiment to determine which instances are realistic and discriminating

As a first step to evaluate the robustness of the solvers, instances of diverse size and structure were generated using the generator in Burke et al. (2010b). 100 instances each were generated with event (lecture) counts 20, 40, 50, 70, 100, 125, 150, 200, and 400; and with 30, 50, 70, 80, and 90% occupancy (time slots − lectures), for a total of 4500 instances. Each instance was run on solvers SACP and TSCS.

Solvers were run on the Monash University Condor pool (Thain et al. 2005), so the running time was normalized based on the processor speed (Mips rating) of each computer used. The results (objective values) obtained by each algorithm for the first 10 normalized minutes at 4000Mips were retained. In almost all cases, the best solution was found within 200 seconds.

On 84% of the synthetic instances, both solvers produced exactly the same objective, often 0 (no penalties), although in every single case they produced different solutions. The behavior of the solvers on the 21 real instances from ITC2007 was very different. Only 14% of the solutions had the same objective value for the real set, and only one solution was perfect. This is especially interesting given that the two codes are markedly different in design and implementation. On the 21 real instances, SACP wins 10 times, TSCS wins 8 times, and there are only 3 ties. This evidence is compelling, but not sufficient to conclude that the instances in S are not relevant. Furthermore, the evidence does not help to understand what is wrong with the instances so S can be improved.

Occupancy is a parameter to g, so a simple difference of means experiment can be used to test whether this parameter may be the culprit. There is no conclusive statistical evidence to support that occupancy has an effect on tie probability. A more sophisticated experiment that can give specific insight into how to change the generator is needed.

3.1.1. Metadata The first step in the experiment is to measure the features (also known as metadata: data about the data). Specifically, we would like to measure characteristics of the problem that are known to be predictive of performance, and compare those measurements on instances from both S and R.

Udine Timetabling captures some important characteristics of timetabling problems in a simple and elegant framework. It was proposed in its specific form for the ITC2007. Thus, compared to other problems, there are relatively few statistical features specific to CTT, beyond those that come from the problem definition itself, whose impact on performance is well understood.

In this scenario, features from related problems (in this case, other variations of timetabling problems) can be used, as well as features from Graph Coloring, which is the core problem most clearly related to Udine timetabling. The graph coloring literature predicts that randomly generated problems using standard univariate distributions are very easy (Culberson 2001). This is evidence that extracting features related to the underlying vertex coloring problem is important.

Next, many features related to the underlying vertex coloring problem were computed, shown by a diverse set of literature sources (see Smith-Miles and Lopes (2012) for a comprehensive review) to be predictive of difficulty, although not necessarily of tie behavior. A total of 32 features were used, and are described in EC.1 and in the Coin-OR distribution of the tools provided in EC.2. The most important features are summarized below, especially those that will appear in illustrations:

- Features related to the underlying Graph Coloring problem for three conflict graphs, produced by: curriculum conflicts; teacher conflicts; and combined teacher and curriculum conflicts.
  - —clustering coefficients (Beyrouthy et al. 2008) and node degrees (Kostuch and Socha 2004).
  - —The number of colors computed by the DSATUR (Wood 1997) heuristic, and the sum of colors (it tries to assign low-numbered colors when possible).
- Features identified in the timetabling literature
  - —Number of students in a course
  - —Number of courses, rooms, lectures per course,
- Features specific to Udine timetabling
  - —Room options for each course
  - —The proportion of lectures that could only be scheduled in one room

Figure 3 A classification tree describes features predictive of ties (non-discriminating), non-ties (discriminating), and real (competition) instances. The output from R has been edited for clarity, and values have been rounded due to sample size. Indentation indicates the level of the tree.

Node	competition	non-ties	ties
1. Root	21	50	50
2. Curriculum Clustering Index Mean≥ 0.6	**21 (100%)**	**0**	**0**
3. Curriculum Clustering Index Mean<0.6	0	50 (50%)	50 (50%)
4. Curriculum Clustering Index Std. Dev.<0.4	0	36 (72%)	14 (28%)
6. Mean Room Options <3.6	0	34 (85%)	6 (15%)
7. Mean Room Options ≥3.6	0	2 (20%)	8 (80%)
5. Curriculum Clustering Index Std. Dev.≥0.4	0	14 (28%)	36 (72%)
8. Proportion of One Room Lectures ≥0.1	0	10 (59%)	7 (41%)
9. Proportion of One Room Lectures <0.1	0	4 (12%)	29 (88%)

Many of the alternative representations of graph conflicts were highly correlated, so we eliminated some of them when analyzing the metadata.

In the discussion below, we report on a single outcome of the experiment, but in every case we repeated it many times and obtained results that are not substantially different.

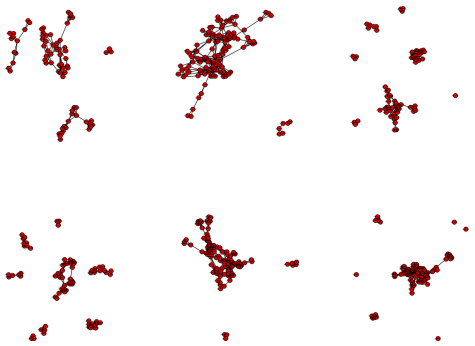
3.2. Insights from the experiment are used to change the generator

An experiment using the `rpart` package from `R` produced classification trees like that of Figure 3. Each rule partitions the set of instances in that node into two subsets. The goal of the algorithm behind the tree is to maximize *purity*, the percentage of instances under a node of the tree that belong to the predominant class. In Figure 3, node **2** has perfect (100%) purity.

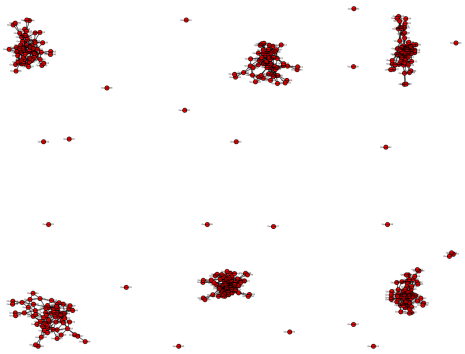
The typical goal of experiments using classification trees is to find the features that will lead to high purity nodes. In those cases, a successful experiment is one that produces nodes of high purity. In contrast, in this research, we first fix the features, then try to confuse the classification algorithm as much as possible by changing the nature of the instances we ask it to classify. In this context, a successful experiment is one in which both real instances R and synthetic instances S are dispersed throughout the tree. This will mean that no existing feature can distinguish real from synthetic instances (so our synthetic instances are real-world-like); and no existing feature can distinguish discriminating from non-discriminating instances (so any observed difference in solver performance cannot be explained by existing features that relate to known results, and we have created the potential for novel insights).

Figure 4 6 randomly chosen conflict graphs from real instances (4a) and from synthetic instances (4b) with approximately the same number of courses. There are more clusters in the conflict graphs of Figure 4a, and the distribution of nodes in each cluster has a higher mean (see rule **2** in Figure 3).

(a) Competition instances



(b) Burke et al. (2010b) generator instances



Assuming that not all R instances are discriminating, it is important to consider both discriminating and non-discriminating instances in S, and it is important to confound the non-discriminating class with the other two. Otherwise, in the attempt to break the separation between S and R, we may end up with a set S that overemphasizes the features that leads one solver to defeat another, or end up with a set S that is not discriminating.

The classification tree rules most successful at separating R from S and separating discriminating from non-discriminating instances provide straightforward guidance that can be used to improve g. Figure 3 makes clear the main problem with this instance set S: the underlying graph coloring problem has a very different structure in S than it does in R. From the resulting classification tree rules, it is clear that there is a problem with the way the generator creates curricula. The tree has pointed out to us that of all the things that could go wrong (room sizes, curricula lengths, enrollments, slack, etc.), the conflict graphs are key, and the remedy is clear: the Curriculum Clustering Index Mean must be raised above 0.6 in the synthetic instances.

Figure 4 illustrates graphically that the distributions from which the graphs in R and S are drawn are different; and that the graphs in Figure 4a are not from a familiar distribution. They are not $G_{n,p}$ graphs (Erdos et al. 1966). One might expect them to be scale-free (graphs whose degree distribution follows a power law), but they are not quite so. A linear regression on a semi-log plot of node degrees

Figure 5 This tree demonstrates improvement through nodes that are not pure. A smaller tree is also sign of success – the partitioning algorithm will stop if it cannot find a rule that will guarantee at least a minimal level of classification improvement in the children relative to its parent.

Node	competition	non-ties	ties
1. Root	21	21	21
2. color sum$\geq$ 1136	1 (10%)	9 (90%)	0 (0%)
3. color sum$<$ 1136	20 (38%)	12 (23%)	21 (39%)
6. slack$<$77.5	3 (11%)	9 (32%)	16 (57%)
8. Curr. and Teac. conflict graph connectivity $\geq$ 0.09	2 (13%)	7 (47%)	6 (40%)
9. Curr. and Teac. conflict graph connectivity $<$ 0.09	1 (8%)	2 (15%)	10 (77%)
7. slack$\geq$77.5	17 (68%)	3 (12%)	5 (20%)

gives a reasonably good fit above degree 2, but there are too few degree one nodes. Even for the range where the power law applies, there is more structure: the graphs connect courses which reveal relationships between departments and disciplines that are more specific than those that can be obtained simply by sampling scale-free graphs. A solver tested against scale-free graphs may produce disappointing results at deployment time, when it turns out that real problems have the structure of a different type of graph coloring problem.

Ultimately, all simple workarounds failed to worsen the separation, so a bootstrapping simulation of course conflicts was employed, which was succesful. The approach, along with a few further iterations of the main loop of Algorithm 1, are described in the electronic companion EC.3 to this paper.

When using the final generator which benefited from several rounds of feedback from the classification tree (version 4), 90% of instances were discriminating (very close to the 84% of the real dataset). These instances were also much harder to separate from real instances. The difficulty in generating non-discriminating instances now meant that we chose to build a classification tree based on 21 instances from each of the three classes. The tree in Figure 5 is typical of the application of the classification tree to this latter dataset.

While improvements can still be made, and there is still high concentration of real instances under node 7, every node except node 2 contains some instances from each class, which may be sufficient when the goal is not to classify (as is traditionally the case with classification trees) but rather to produce a data set which is hard to separate. Further details about a simple statistic that can be used to assess how poor the separation is can be found in Section 4.

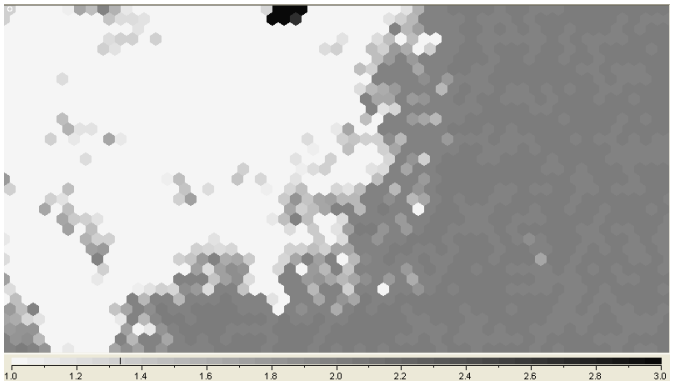
3.3. Validating the improved instance generator

The process described in the previous sections has shown how an initial synthetic instance generator can be modified, via feedback about the differences between the synthetic instances and real instances, so that the improved generator creates instances that are more similar to real instances, capable of discriminating solvers, and not biased by any well-known structure. To double-check the output from the classification trees, which is a supervised learning approach whose outcome is mostly quantitative, we use an unsupervised learning technique whose output is mostly visual and qualitative. To achieve visualization of a high-dimensional feature space, we use a self-organizing map (Kohonen 2002), which aims to preserve as much as possible the high-dimensional topology of the instances when projecting them onto a two-dimensional plane.

Figure 6 shows the location of the three classes of instances across this two-dimensional projection of the feature space. We refer the reader to Smith-Miles and Lopes (2011) for details of this implementation. Instances are represented as points in the map, with two instances located close to each other if they are close in Euclidean distance in the high-dimensional feature space. The small set of real-world Udine instances are all located in the top-center of the map (shown as black on the map), and are clearly very similar to each other. The original synthetic instances from the generator of Burke et al. (2010b) (shown as gray) are not in the same region as the real-world instances. The instances from the generator that we have modified to be more "real-world-like" (shown as white) surround the Udine instances and are therefore quite similar based on their features, but are more diverse. The distribution of each feature across this instance feature space can be inspected to try to visually understand the critical differences between the synthetic and real instances, as we have shown in our related work (Smith-Miles and Lopes 2011) which utilized the instances generated by the methodology described in this paper.

It should be noted that no information about the class of instance was used to project the instances, only features of the timetabling problem and the underlying graph coloring problem. Yet the three classes of instances are clearly seen as quite distinguishable in this instance space. From here, the footprint of each algorithm can also be inspected by super-imposing algorithm performance on instances across the feature space (see Smith-Miles and Lopes (2011) for further details, beyond the scope of this paper).

Figure 6 Visualization of three classes of instances in feature space (projected to 2-dimensions using a self-organizing map)



3.4. Addressing the example claim

Recall the experimental claim for the example:

After 10 minutes of running time, SACP produces objective values at least as good as those from TSCS on instances like those from the ITC2007 competition.

In the set of 21 real instances, there are 3 ties; SACP wins 10 times; and TSCS wins 8 times. So for this small set of ITC2007 instances we can see that for only 62% of the instances the claim is supported. If the dataset produced by the original generator was used to support this claim though, there is a danger that invalid conclusions will be drawn. Only 810 instances out of 5000 were discriminating in that set. If we consider only performance on the 810 discriminating instances, TSCS wins on only 67 instances, and so we could argue that the claim is supported for $4933/5000 = 98.6\%$ of the instances. However, we have already demonstrated that the original synthetic instances are not very similar to those from the ITC2007 competition. The instance set is not applicable to support any claim involving competition-like instances, even though the generator has been designed to produce instances describing the same real-world problem. For any algorithm we are likely to be able to find some set of instances where superior performance can be demonstrated (Wolpert and Macready 1997), but we must ask whether these instances are relevant to the claim, and whether the research findings are interesting.

Using the set of instances S' from the improved instance generator, we find that it is difficult to distinguish between the two solvers on competition-like instances, just as it is on the small set of 21 real competition instances. Only in 56% of instances in S' do we find that SACP is at least as good as TSCS. Our new synthetic dataset has been demonstrated to be applicable to a claim about performance on

competition-like instances, since we cannot easily separate S' from the instances in R (noting that the competition instances are based on a real university) using the feature set. We also cannot easily separate the discriminating and non-discriminating instances using the feature set, and so if we do find a subset of instances where one solver outperforms the other, it won't be due to any obvious properties, and we have the potential for novel insights.

So, on these competition-like instances, we cannot support the claim that the performance of SACP is at least as good as TSCS, which is the conclusion we would have reached based on the original synthetic generator. Some further analysis is needed to refine the experimental claim and is provided in EC.4.

4. The Expected Applicability Coefficient

It is important to produce a meaningful summary statistic of the applicability of an instance set based on the outcomes of the resulting classification tree. Not only can this help to design generators that can automatically tune themselves to instance sets, but it also provides a convenient way for researchers and referees to evaluate whether a dataset is sufficiently applicable for the purposes of a study. For this purpose, this research proposes the Expected Applicability Coefficient (EAC), a statistic derived from existing measures of the quality of classification trees.

Summarizing a classification tree with a simple statistic is admittedly fraught with perils, since the tree is a result of the distribution of the instances used to train it. The instances of a real problem should not be expected to be distributed uniformly over feature space, since when working through real problems, people find ingenious ways to adapt business processes to the constraints at hand. In the absence of a reasonable prior, the EAC should be used with care.

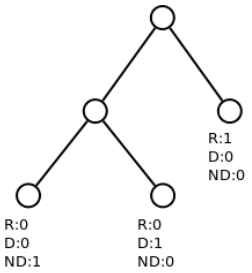
The only prior available on the distribution of features over $\mathcal{F}$ is the sample distribution of the instances in R. Since R is usually small, special care must be taken to avoid over-fitting. Thus, in most cases it is probably a folly to continue changing the generator too many times until a perfect failure to separate emerges. How much improvement is "good enough" requires further research and will emerge over time. Even with these caveats, we still find the EAC (and its cousin, the ERC, which measures separability only along the realistic axis) useful.

To measure the quality of separation at any node, the concept of *node impurity* is commonly used in the classification tree literature. Node impurity is a measure of the relative frequency of different classes

Figure 7 R, D, and ND are the classes in C. For each node v the numbers in Figures 7a and 7b are the p_{iv}. The ideal tree has no useful separating rule; the worst possible tree perfectly separates three classes with only three leaves.

(a) The ideal tree. The classification tree algorithm cannot find any good separating rules.

(b) The worst possible trees are any permutation of a three leaf tree, where each leaf is perfectly pure. One rule perfectly separates one class from the other two, and a second rule perfectly separates the two remaining classes.



within a node, but it a more ambiguous concept that measuring node purity (the percentage of the node corresponding to the dominant class). If a node contains only instances of one class, then its impurity is zero. Its impurity is maximal when under a node there are the same number of instances of each class. Any function of the frequency of classes in the node will do as a measure of impurity provided it satisfies some simple properties (Breiman 1984). The functions should be concave; attain their minima when the probability of any class is 1, and therefore the probability of the other classes is 0; obtain their maxima when all classes are equally likely. The most prevalent impurity measure used in the research literature is the Gini coefficient (not to be confused with the related but distinct measure of concentration of wealth). A simple adaptation of this concept gives a good sample statistic of the applicability of the instance set.

The scaled Gini coefficient Given a set C of classes, the Gini coefficient of a node v within a classification tree is calculated from the relative frequencies p_{iv} of class i in node v, $\forall i \in C$:

$$\text{Gini}(v) = \sum_{(i,j)\in C\times C | i\neq j} p_{iv}p_{jv} = 1 - \sum_{i\in C} p_{iv}^2 \tag{1}$$

For the purposes of this research, the ideal tree and the worst possible tree are illustrated in Figure 7. It is desirable that the tree in Figure 7b has a coefficient of 0, while the tree in Figure 7a has a coefficient of 1. A computation that will achieve this result and is derived from the existing literature is to compute the average Gini coefficient (Equation 1) for each leaf node and divide that average by the normalizing constant 2/3. The constant 2/3 is the highest possible Gini coefficient of a node when three classes are

Table 2 Summary of the EAC and ERC for different generators. The last column (ERC) is the quantity analogous to the EAC (the mean gini coefficient of leaf nodes normalized by 1/2) computed on a two-class tree that separates only Real from Synthetic instances.

Parameter Set	Description	EAC	ERC
Original (Sec. 3.1)	Version 1: Original generator (Figure 4)	0.34	0
	Version 2: added graph bootstrapping	0.51	0.37
	Version 3: added weighed course popularities	0.55	0.52
	Version 4: fixed course and room sampling bug from Version 3	0.52	0.5
Matched Events and Slack (EC.3.1)	Version 1: Original generator (Figure 4)	0.33	0
	Version 4	0.61	0.52

present. The average of these coefficients, over a sample n of subsets $S_i \subset S$ is the EAC of S. If $L(S_i)$ is the set of leaves from a classification tree produced from a sample $S_i \subset S$ and the experiment is run n times, then the sample EAC of S is:

$$\mathrm{EAC}(S) = \frac{3}{2} \frac{\sum_{i=1}^{n} \frac{\sum_{v \in L(S_i)} \mathrm{Gini}(v)}{|L(S_i)|}}{n} \tag{2}$$

Similarly, the Expected Realism Coefficient (ERC) measures the lack of separation between only two classes: S and R. In this case, the normalizing constant is 1/2. Instance sets produced during this research had EAC between 0.25 and 0.75, depending on the generator, the choice of features used, and the parameters used. It is possible to artificially inflate the EAC by careful selection of features, and the results will also depend on the parameters used for the construction of the classification tree. Thus, the EAC should accompany the set of features used to compute it, and clearly specify the classification tree parameters (default parameters used by the R package `rpart` for the results in this paper). Using a weakly correlated comprehensive set of features (see EC.1), the sample EACs and ERCs of the instance sets corresponding to each version of the generator with a sample size of 1000 are reported in Table 2.

Table 2 demonstrates the dangers of not considering realism and discrimination ability together. Although changing the parameters of the original generator to match the sample distribution of the real data produced 63% differentiation, those instances were very easy to separate from the real instances (EAC 0.34, ERC 0). Thus, comparisons on solver performance using that generator would have a serious flaw, and solvers built to perform well on those instances might perform poorly at deployment time.

5. Conclusions

This research presents the first concrete procedure for evaluating the applicability, defined in specific terms, of an instance set and instance generator to claims involving the comparison of solver performance

on complex branch problems. It is the first procedure built to make concrete some of the guidelines for instance generation suggested by many authors in published research (Hall and Posner 2010, Hooker 1995, Johnson 2002, Moret 2002, Anderson 2002, Bartz-Beielstein et al. 2010). The research shows that building on a simple notion – that an instance set is similar to another if the two cannot be separated using features known in the literature to be important – can lead to significant improvements in instance generators. It also provides a way to quantify those improvements in a straightforward manner. Furthermore, when generators have shortcomings, this procedure can specifically identify them.

By applying these techniques, researchers can avoid empirical, inefficient attempts to change instance generators without clear guidance as to what improvements would have the greatest impact. Better generators, in turn, will lead to solvers that perform more predictably at deployment time, and to more precise and authoritative claims regarding methodological advances. The procedure is especially useful on complex problems which contain many nuances and are related to a variety of underlying problems, which we refer to as branch problems.

To illustrate the proposed procedure, it has been applied to the Udine Timetabling problem. The paper demonstrates significant progress in instance quality that is measurable analytically and is also readily verified in practice, especially since the framework for analysis is Open Source available through Coin-OR.

The techniques in this paper can also be applied to core problems. However, the case for such an approach is weaker when the subject of study is a core problem for the following three reasons: 1) core problems tend to appear in many branch problems, so that almost any reasonable instance structure is likely to be found in some application; 2) core problems are easier to abstract, and thus lend themselves more readily to natural and relevant analytical results, which generally stand on their own as a research contribution even when the computational study is lacking; 3) core problems tend to be interesting subjects of mathematical study whether or not they have practical importance. Their mathematical beauty is sufficient.

In contrast, computational results tend to be the most useful outcomes from studies of branch problems, even when analytical results are present, because the interactions between conflicting requirements in branch problems tend to be more useful predictors of performance than worst-case results. Satisfying some requirements in branch problems involves potentially intractable procedures like sequencing, partitioning or coloring, while others can be optimized by tractable procedures like sorts, assignments, or shortest

paths. Because branch problem definitions usually include a potentially intractable problem, they tend to be trivially intractable in the worst case. However, better theoretical guidance on solver design can be obtained by studying which of the underlying core problems in the branch problem is most salient in instances of interest. If the instances that are likely to appear in practice are of one kind, it is at least uninformative to compare solvers using synthetic instances of another kind.

These techniques can help strengthen a claim often made in the literature but usually left unjustified: that a specific model or algorithm performs better than another one on real-world-like instances. These techniques should not be used to help evaluate the broader claim that a model was the correct one to apply to a situation. They are simply a first step towards making concrete the abstract guidelines offered by previous research into instance generation, and to providing a useful framework for evaluating claims involving complex, real problems.

The validation techniques here can also be adapted in future research to obtain insights into the actual relative performance of algorithms (instead of only on the quality of the instances); they can be incorporated into meta-solvers for branch problems; and new validation procedures can be devised which compute summary statistics of applicability that explicitly take account of priors regarding instance distributions. The techniques here can also be generalized to other fields; and they could also borrow from other fields, especially through classifiers specifically designed for this problem. These new classifiers should aim to improve separation when compared to classification trees, while maintaining the simplicity of the feedback, perhaps by using a limited family of separation manifolds.

While this research has established a framework that guides manual improvement of an existing instance generator, we are interested in future research to explore replacing the initial generator with a simulation-based approach with tunable parameters. More parameters enable a more formal experimental framework such as that suggested by Hooker (1995) and would allow for automatically creating discriminating instances. A simulation-based generator will also allow an entirely new set of questions to be answered involving the impact that business processes may have on the hardness of a problem.

Further research also must probe more into the impact of the feature set on the effectivess of the procedure. We have made some assumptions that a well-developed literature is available that provides guidance

about which statistical properties of instances are relevant to questions about hardness of instances (Smith-Miles and Lopes 2012), and that we can easily measure the kinds of properties that might distinguish real-world from random instances, or easy from hard instances. We have assumed that our chosen feature set is so comprehensive and well-researched that all known results are captured by this set of features. If we do not start with a suitable feature set though, then we cannot expect the relationship between instances that we observe in the feature space to be necessarily accurate or enlightening. Additionally, we need to ensure that the chosen features translate into actions to modify the instance generator. There is little point gaining insight into how a particular feature of the instances should be varied if the instance generator has no control over the feature. Clearly, the interplay between the feature set and the instance generator parameters is a critical one to control, and as we consider generalizations of the proposed procedure to other problem domains, this will need further consideration.

Another important question is how to validate instances that are interesting, but in some way are not like those from R. The typical case is where it is desirable to test solvers against instances that are larger than the ones in R, while otherwise preserving the structures of those instances. The general notion of interpolating feature measurements seems useful in that case, but true progress will require significant further research.

This research also demonstrates the power of a new, open source approach to computational experimentation. This research was enabled by open source, and aims through open source to enable other research and practice (see EC.2 in the electronic companion to this paper). Open source can go a long way toward addressing other components of the experimental frameworks recently proposed by several authors, especially reproducibility. Another issue involving reproducibility and open source is the absolute need to make the evaluation of instance set quality as simple and unobtrusive a part of the research process as possible. It is unreasonable to expect researchers whose interests lie elsewhere to spend an inordinate amount of effort crafting instances. At the same time, if the literature is to contain their deepest expertise, it is imperative to encourage researchers to use the best instance sets possible in their research. As we ponder future enhancements to the procedures reported here, the compromise between power and tractability needs to be considered.

Acknowledgment

We are grateful for the assistance of the authors of the two solvers (Tomás Müller, and Koji Nonobe) and the instance generator (Edmund Burke, Jakub Mareček), all of which were either available as open source or whose source was made available to us. We also thank the two anonymous reviewers and Area Editor whose valuable insights and suggestions improved the clarity of this paper.

References

Achlioptas, D., A. Naor, Y. Peres. 2005. Rigorous location of phase transitions in hard optimization problems. *Nature* **435**(7043) 759–764.

Anderson, R. 2002. The role of experiment in the theory of algorithms. *DIMACS Series in Discrete Mathematics and Theoretical Computer Science* **59** 191.

Ariyawansa, KA, A.J. Felt. 2004. On a new collection of stochastic linear programming test problems. *INFORMS Journal on Computing* **16**(3) 291–299.

Bartz-Beielstein, T., Marco Chiarandini, Luis Paquete, M. M. Preuss, eds. 2010. *Experimental methods for the analysis of optimization algorithms*. Springer.

Bartz-Beielstein, T., K.E. Parsopoulos, M.N. Vrahatis. 2004. Design and analysis of optimization algorithms using computational statistics. *Applied Numerical Analysis & Computational Mathematics* **1**(2) 413–433.

Beasley, JE. 1990. OR-Library: Distributing test problems by electronic mail. *Journal of the Operational Research Society* **41**(11) 1069–1072.

Beyrouthy, C., E.K. Burke, B. McCollum, P. McMullan, A.J. Parkes. 2008. Enrollment generators, clustering and chromatic numbers. *Proceedings of the 7th International Conference on the Practice and Theory of Automated Timetabling (PATAT 2008), Montreal, Canada*.

Breiman, L. 1984. *Classification and regression trees*. Chapman & Hall/CRC.

Brélaz, Daniel. 1979. New methods to color the vertices of a graph. *Commun. ACM* **22** 251–256.

Burke, Edmund K., Jakub Marecek, Andrew J. Parkes, Hana Rudová. 2010a. Decomposition, reformulation, and diving in university course timetabling. *Computers & Operations Research* **37**(3) 582–597.

Burke, Edmund K., Jakub Marecek, Andrew J. Parkes, Hana Rudová. 2010b. A supernodal formulation of vertex colouring with applications in course timetabling. *Annals of Operations Research* **179**(1) 105–130. URL `http://www.maths.ed.ac.uk/~jmarecek/timetabling/generator/`.

COIN-OR. 2000. *Computational Infrastructure for Operations Research.* URL `http://www.coin-or.org`.

Corne, D., A. Reynolds. 2011. Optimisation and generalisation: Footprints in instance space. *Parallel Problem Solving from Nature–PPSN XI* **6238** 22–31.

Culberson, J. 2001. Hidden solutions, tell-tales, heuristics and anti-heuristics. *The IJCAI-01 Workshop on Empirical Methods in Artificial Intelligence, edited by H. Hoos and T. Stuëtzle.* 9–14.

Erdos, P., A. Rényi, VT Sós. 1966. On a problem of graph theory. *Studia Sci. Math. Hungar* **1**(215-235) 76.

Gaspero, Luca Di, Barry McCollum, Andrea Schaerf. 2007. The second international timetabling competition (ITC-2007): curriculum-based course timetabling (Track 3). Tech. rep., University of Udine. URL `http://www.cs.qub.ac.uk/itc2007/curriculmcourse/report/curriculumtechreport.pdf`.

Gehrke, Johannes. 2003. The handbook of data mining. Lawrence Erlbaum, 3–24.

Hall, N. G, M. E Posner. 2001. Generating experimental data for computational testing with machine scheduling applications. *Operations Research* **49**(6) 854–865.

Hall, N. G., M. E. Posner. 2007. Performance prediction and preselection for optimization and heuristic solution procedures. *Operations Research* **55**(4) 703–716.

Hall, Nicholas, Marc Posner. 2010. *The Generation of Experimental Data for Computational Testing in Optimization*, chap. 4. Springer, 69–98.

Hill, R.R., C.H. Reilly. 2000. The effects of coefficient correlation structure in two-dimensional knapsack problems on solution procedure performance. *Management Science* **46**(2) 302–317.

Hooker, J.N. 1995. Testing heuristics: We have it all wrong. *Journal of Heuristics* **1**(1) 33–42.

Johnson, D.S. 2002. A theoretician's guide to the experimental analysis of algorithms. *American Mathematical Society* **220**(5-6) 215–250.

Koehler, G.J. 2007. Conditions that obviate the no-free-lunch theorems for optimization. *INFORMS Journal on Computing* **19**(2) 273.

Kohonen, T. 2002. The self-organizing map. *Proceedings of the IEEE* **78**(9) 1464–1480.

Kostuch, P., K. Socha. 2004. Hardness prediction for the university course timetabling problem. *Lecture notes in computer science* **3004** 135–144.

Leyton-Brown, K., E. Nudelman, Y. Shoham. 2009. Empirical hardness models: Methodology and a case study on combinatorial auctions. *Journal of the ACM (JACM)* **56**(4) 1–52.

Lopes, Leo. 2010. The O.R. tree. *OR/MS Today* **37**(5) 18–20.

McGeoch, C.C., MA Amherst. 2002. A bibliography of algorithm experimentation. *DIMACS Series in Discrete Mathematics and Theoretical Computer Science* **59** 251.

Monasson, R., R. Zecchina, S. Kirkpatrick, B. Selman, L. Troyansky. 1999. Determining computational complexity from characteristic 'phase transitions'. *Nature* **400**(6740) 133–137.

Moret, B.M.E. 2002. Towards a discipline of experimental algorithmics. *DIMACS Series in Discrete Mathematics and Theoretical Computer Science* **59** 197.

Müller, T. 2009. Itc2007 solver description: a hybrid approach. *Annals of Operations Research* **172**(1) 429–446.

Nonobe, K., T. Ibaraki. 2001. An improved tabu search method for the weighted constraint satisfaction problem. *Infor-Information Systems and Operational Research* **39**(2) 131–151.

Quinlan, J.R. 2003. *C4.5: programs for machine learning*. Morgan Kaufmann.

R Development Core Team. 2010. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing. URL `http://www.R-project.org`.

Reilly, Charles H. 2009. Synthetic optimization problem generation: Show us the correlations! *INFORMS J. on Computing* **21**(3) 458–467.

Shawe-Taylor, J., N. Cristianini. 2000. *Support Vector Machines*. Cambridge University Press.

Smith-Miles, K., T.T. Tan. 2012. Measuring algorithm footprints in instance space. *IEEE Congress on*. IEEE, 1–8.

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

Smith-Miles, Kate, Leo Lopes. 2011. Generalising algorithm performance in instance space: A timetabling case study. *Lecture Notes in Computer Science* **6683** 524–539.

Smith-Miles, Kate, Leonardo Lopes. 2012. Measuring instance difficulty for combinatorial optimization problems. *Computers and Operations Research* **39**(5) 875–889.

Taillard, E.D. 1995. Comparison of iterative searches for the quadratic assignment problem. *Loc. Sci.* **3**(2) 87–105.

Thain, Douglas, Todd Tannenbaum, Miron Livny. 2005. Distributed computing in practice: the condor experience. *Concurrency - Practice and Experience* **17**(2-4) 323–356.

Tuson, AL, SA Harrison. 2005. Problem difficulty of real instances of convoy planning. *Journal of the Operational Research Society* 763–775.

Wolpert, DH, WG Macready. 1997. No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation* **1**(1) 67–82.

Wood, D.R. 1997. An algorithm for finding a maximum clique in a graph. *Operations Research Letters* **21**(5) 211–217.

Leo Lopes is a Senior Operations Research Specialist at SAS. His main research interest is helping people build and use realistic models efficiently.

Kate Smith-Miles is Professor and Head of the School of Mathematical Sciences at Monash University in Melbourne, Australia. Her research interests include combinatorial optimization, data mining, neural networks, and mathematical modeling of problems ranging from neuroscience to economic applications.

This page is intentionally blank. Proper e-companion title page, with INFORMS branding and exact metadata of the main paper, will be produced by the INFORMS office when the issue is being assembled.

Additional Details

EC.1. Features for the Udine Timetabling problem

Features used in the summaries in Table 2 are marked with a *

- DSATUR outputs:
  - —Estimate for the number of colors *
  - —Sum of color values over all nodes *
- From each of: the curriculum conflict graph; the teacher conflict graph; and the combined conflict graph:
  - —The density (connectivity) of the graph *
  - —Clustering coefficient.
  - —Mean and standard deviation of the distribution of node clustering indexes (density of the subgraph induced by each node) *
  - —Mean and standard deviation of the event degree distribution:
    - Weighted by enrollment count
    - Unweighted
- Slack (total seats offered - total seats requested by all courses) *
- Mean and standard deviation of the distribution of course enrollments.
- Mean and standard deviation of the distribution of the number of rooms into which a course may fit.
- Number of events that will only fit in one room *

EC.2. Open source code

This research offers a methodology to address the issue of instance applicability more precisely: when an instance set has shortcomings, the procedures here can identify them; the applicability of the instance set can be summarized using a simple statistic (presented in Section 4 of the paper); and the entire process can be undertaken in an iterative manner with the goals of the process clearly defined. These three requirements—providing actionable feedback; establishing an overall measure of instance set quality; and doing so without adding significant complexity to the solver creation process—are all crucial to enabling significant improvements in experimental OR within the reality of how solvers are created today. Satisfying them drives the outcomes reported in this paper.

The operations performed in Section 3 can be formalized into a heuristic. When a generator with robust controls that can be automatically set is available, the heuristic can to a great extent be automated (this is the topic of current research on which we will report at a later point). In this paper, the role of "guiding the search" is performed by the operator. Fundamentally, however, the algorithm's major steps are the same:

1. Collect a seed set R of real instances
2. Enumerate a set of relevant features
3. Generate a diverse set of instances
4. Run all solvers on all instances; classify the instances into three groups: real (R); synthetic differentiating (SD) ; and synthetic non-differentiating (SN).
5. Repeat:
   (a) Sample $|R|$ instances each from SD and SN.
   (b) Run a classification tree on the sampled instances and annotate patterns that consistently distinguish any of the three sets from the other two.
   (c) If no rules consistently emerge, or if they do emerge but they separate only a small number of instances of any class, stop.

In the implementation, the experiment input consists of comma-separated files containing the metadata for each instance in each row, and the features and algorithm performance in each column. The implementation is coded in R (R Development Core Team 2010) and is available from the Coin-OR page associated with this paper along with detailed technical explanations on how to replicate the experiment.

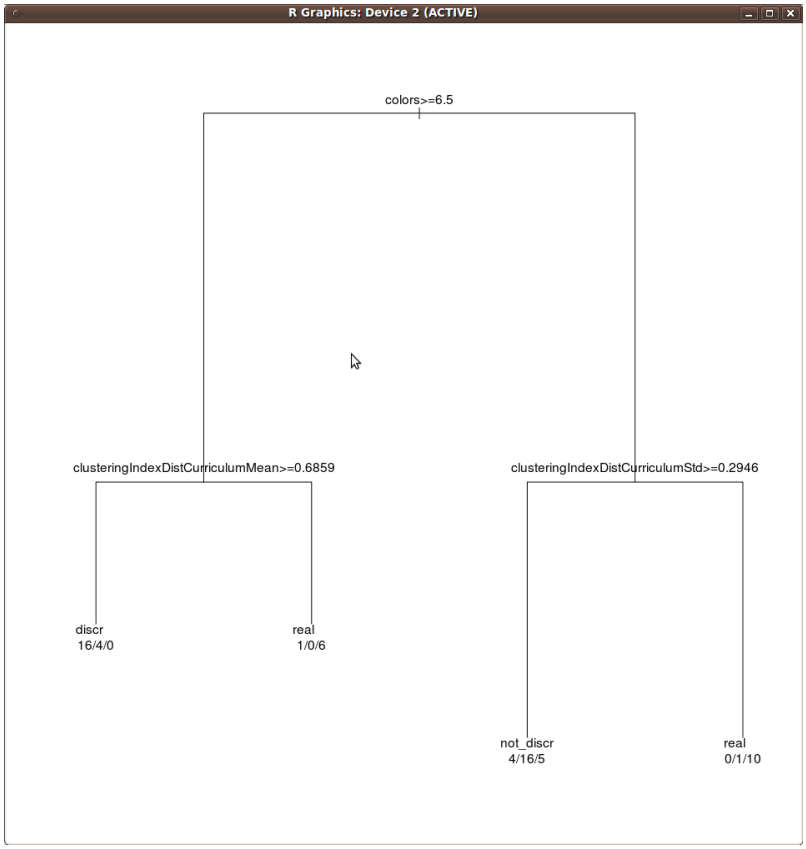
EC.2.1. Web addresses

- For the generator: http://code.google.com/p/udinettgen/
- For the R Applicable Instances package: https://projects.coin-or.org/ApplicableInstances/

EC.2.2. The Coin-OR package

Below is a sample session using the library. There are six steps:

Figure EC.1 An example tree as produced by the routines in the library



1. Using your favorite program, create two comma-separated (csv) files: one for the real instances (in this example, *real.csv*), and one for the synthetic instances (in this example, *synthetic.csv*). The csv file should have each instance in a row, and each feature in a column. The first column should be the name of the instance.

2. Start R and load the `applicableInstances` library

3. Pass the locations of the csv files, the names of the columns containing solver performance data, and optionally, an indifference level to the library and get back a data frame.

4. Define a list of features to base the classification on.

5. Compute an estimate of the EAC or ERC

6. Obtain and plot trees. The tree produced by this command is in Figure EC.1.

More details can be found at the project's homepage at Coin-OR (https://projects.coin-or.org/ApplicableInstances/). There are a few further options in the library that are not documented here. Over time there will be improvements to the library, both in terms of methodology and features. However, the main functionality described in this paper will not change. It is designed to be as straightforward to

use as possible. The goal is to keep the instance validation time to a minimum, so that researchers can focus on building better algorithms and writing insightful papers, and practitioners can focus on avoiding deployment surprises, instead of spending a lot of time working with the package.

EC.3. Modifications to the generator based on classification tree feedback

After the classification tree helped to identify that the competition instances and the synthetic instances have a different underlying graph structure (see Figure 4, we decided to modify the generator by bootstrapping the graphs. Each sub-graph in each of the 21 instances in R is collected into a single large super-graph of 141 connected sub-graphs. Then, when generating new instances, sub-graphs are sampled from this super-graph. While this represents some loss of completeness of the generator, it provides an important opportunity to integrate known distributions of real-world instances into the improved instance generation process.

An intermediary generator using this approach (version 2 of the generator) produced a more confounded tree (omitted) that suggested a subsequent change: instead of sampling uniformly from the super-graph, the probability of selecting each sub-graph should be chosen to be the mean relative popularity of the courses in the super-graph. This produced a more confused (better) classification tree (version 3 of the generator). Courses are added as needed by randomly picking adjacent vertices along each sub-graph until the required number of events is inserted into the instance, or the connected sub-graph is exhausted. When it is exhausted, a new connected sub-graph is sampled. This approach preserves the relationships between subjects in the original graphs, while producing sufficient diversity in the new instances, due to the large number of sub-graphs.

This version of the generator often created invalid instances, because of the relationship between room sizes and course enrollments. Fixing that issue resulted in version 4 of the generator. This version slightly worsened the outcomes from the experiment (improved the quality of the separation). At the end of this step, trees similar to those in Figure EC.2 were produced. On this tree, separation is far less successful than it is in the instances pictured in Figure 4, which is sign of progress.

The tree in Figure EC.2 tells us that the number of courses that can only fit into one room is a primary differentiator between real and synthetic instances (nodes 2 and 3), especially when combined with "color

Figure EC.2 A classification tree describes features predictive of ties, no ties, and competition instances. The output from R has been edited for clarity, and values have been rounded due to sample size limitations.

Node	competition	non-ties	ties
1. Root	21	50	50
2. one room events< 1.5	17 (81%)	3 (6%)	7 (14%)
4. color sum≥495	16 (76%)	3 (6%)	0 (0%)
5. color sum<495	1 (5%)	0 (0%)	7 (14%)
3. one room events≥1.5	4 (19%)	47 (94%)	43 (86%)
6. event connectivity<0.16	3 (14%)	40 (80%)	18 (36%)
8. one room events≥10.5	0 (0%)	26 (52%)	5 (10%)
9. one room events<10.5	3 (14%)	14 (28%)	13 (26%)
...	...	...	...

sum". Color sum is the sum over all nodes of the colors assigned to each node. It is obtained by running an implementation of the DSATUR heuristic (Wood 1997). This heuristic assigns colors to the graph in a greedy way, then backtracks and attempts to remove colors, giving highest priority to higher indexed colors. The ratio of color sum to the number of nodes was also used in this experiment, but seldom appeared in any trees, which indicates that the actual size of the instances are an important factor. The feature "One room events" as a separator indicates that slack (defined as total seats offered - total seats requested by all courses) should be looked at more carefully as well.

EC.3.1. A few further iterations

With the feedback from this tree, a histogram of the number of events in each instance was produced, and a new batch of instances whose number of events matched the distribution in the original data were generated. In hindsight, it seems obvious that this should have been done first. However, in preliminary studies with the original data, before employing classification trees, instances with event counts similar to the real data were no more likely to be discriminating than instances of a different size. This observation was confirmed when the original and new generators were used with the new parameters: while the performance of the generator incorporating feedback from the trees improved significantly, the performance of the original generator actually was slightly worse along the realism dimension.

Not all changes resulted in an immediate improvement. Sometimes a change to the generator would cause the features chosen close to the root to change, requiring yet another iteration or two until significant improvements were found.

To make sure that the effects observed were really a result of changes made with the benefit of feedback from the classification trees, both the original generator and the revised generator were used with the new parameters. Using only the original generator (version 1) but matching the slack and event distributions carefully, 63% of synthetic instances were discriminating (versus 16% with the previous parameters and 84% in the real dataset), a significant improvement. However, the separation between real and synthetic instances was still relatively easy for the classification tree.

EC.4. Further analysis of solver performance

SACP crashed on 514 instances (all but one of which was proven infeasible using integer programming), while TSCS did not crash on any instances. On many of the instances on which SACP crashed, TSCS produced good results, although with penalties for violating hard constraints, while the steps involved in SACP clearly precluded it from performing well if an initial feasible solution cannot be found (Müller 2009). Thus, we can conclude that, at least using the setup in this paper, TSCS is the more reliable solver, and our methodology has generated some instances where the strengths and weaknesses of solvers can be discriminated.

Figure EC.3 provides more insight than a simple comparison of means. It suggests that TSCS beats SACP often, although not by much, and that when SACP wins (assuming it doesn't crash), it wins more convincingly. These are very different claims to those that would have been supportable with instances from the original generator.

Figure EC.3 A histogram of the difference in performance between solvers for each successfully-run instance. The horizontal axis is the difference in objective values between TSCS and SACP algorithms. A positive value indicates that SACP won. There was also an instance, not used to prevent skewing the plot, where the difference was 1099.