Mathematical Programming Computation (2020) 12:389–415
https://doi.org/10.1007/s12532-019-00170-6

FULL LENGTH PAPER

Generation techniques for linear programming instances with controllable properties

Simon Bowly[1] · Kate Smith-Miles[1] · Davaatseren Baatar[2] · Hans Mittelmann[3]

Received: 20 April 2017 / Accepted: 5 August 2019 / Published online: 17 August 2019

Abstract
This paper addresses the problem of generating synthetic test cases for experimentation in linear programming. We propose a method which maps instance generation and instance space search to an alternative encoded space. This allows us to develop a generator for feasible bounded linear programming instances with controllable properties. We show that this method is capable of generating any feasible bounded linear program, and that parameterised generators and search algorithms using this approach generate only feasible bounded instances. Our results demonstrate that controlled generation and instance space search using this method achieves feature diversity more effectively than using a direct representation.

Keywords Linear programming · Instance generation · Controllable properties · Encoded space

Mathematics Subject Classification 90C05 · 68W40 · 90-08

This research is funded by the Australian Research Council under Australian Laureate Fellowship FL140100012.

✉ Simon Bowly
s.bowly@unimelb.edu.au

Kate Smith-Miles
smith-miles@unimelb.edu.au

Davaatseren Baatar
davaatseren.baatar@monash.edu

Hans Mittelmann
mittelmann@asu.edu

1 School of Mathematics and Statistics, University of Melbourne, Parkville, VIC 3010, Australia

2 School of Mathematical Sciences, Monash University, Clayton, VIC 3800, Australia

3 School of Mathematical and Statistical Sciences, Arizona State University, Tempe, AZ 85287-1804, USA

1 Introduction

The perceived success of algorithms based on empirical performance analysis, in optimisation as in many other fields, is highly dependent on the quality of test instances available. Researchers draw conclusions about the strengths and weaknesses of algorithms based on these test instances, and we must ensure they are unbiased, representative, and diverse in their measurable features or properties. Many benchmark test instances are not suitable for this purpose, since they are often based on a limited set of real-world problems, or have been inherited from earlier studies that may now be obsolete [2,15]. MIPLIB [20] is a good example of a collection of test instances for mixed integer programming (MIP) problems that are refreshed periodically, acknowledging that the difficulty of test instances needs to keep pace with advances in algorithm development. The approach for augmenting and updating MIPLIB, however, is to call for submissions of interesting, challenging, and real-world test instances. While the most recent update, MIPLIB 2017 [10], aimed to maximise diversity and balancedness with respect to instance features in the final test set, the result is limited by the diversity of the submitted problems.

Synthetic test instance generators provide an alternative source of data for experimentation in optimisation, however, their design must be carefully considered. It is well known that simple random generation approaches tend to produce instances which have predictable characteristics [1] and which are not very diverse in measured features [13,26]. Experimental hypothesis testing requires data where test parameters are appropriately varied and other influences are randomised [11,15,23]. Consequently, Hooker [14] advocated for the use of highly parameterised generators to produce appropriately controlled data for experimentation.

We should clarify the goals of instance generation techniques in this paper before proceeding. Firstly, we do not propose replacing real-world benchmarking sets with generated instances. When tuning algorithms for a specific industrial application, it makes sense to use a library of instances tailored to that application. If that set is representative of the problems seen in practice, benchmarking and tuning against that set is likely to yield useful results and a reasonable picture of practical performance. Generated instances are more applicable to exploratory analysis and algorithm stress-testing, where we are interested in testing performance and finding limitations of algorithms on previously unseen instances. Secondly, we aim to develop a highly parameterised generator, which gives control over features of interest. It is presented in opposition to a 'naïve' approach, which generates a distribution of random instances.

For various combinatorial optimisation problems—quadratic assignment [8], multidimensional knapsack [12], graph colouring [7] and Boolean satisfiability [1]—there has been significant attention paid to producing controllable instance generators. Each of these studies has shown that careful generator design is absolutely necessary for experimental results to be valid and generalisable [13]. Given the proliferation of portfolio-based and automatically configured solvers [17,30], and the need for representative test instances in the tuning of such strategies [16,25], it is especially important to develop methods to generate test data with controlled variation of all features of interest.

Existing generators for linear and integer programming instances may not satisfy these requirements. The NETGEN generator [19] and its successor MNETGEN produce parameterised linear programming (LP) instances, but are targeted specifically at multi-commodity flow, transport and assignment problems. The parameters used are thus appropriate to the underlying network, not the LP feasible set or solution.

Todd [28] investigates properties of randomly generated feasible polyhedra. The formulation used to guarantee feasibility is similar to what we develop in later sections of this paper, however, there is no mechanism given to control the features we aim to vary in this work. Pilcher and Rardin [24] define a generator for pure integer programming problems with a known partial polytope by introducing random cuts. However, the implementation is restricted to travelling salesman problems and does not consider the relaxation solution or structural features explicitly. Without the ability to vary features of interest, the utility of these generators for experimentation would be limited to specific problem domains.

It is not always easy to select parameters of an instance generator so that properties of interest are suitably varied. Certainly, some properties can usually be explicitly controlled through the generation process, such as the density of a graph. Other properties will be harder to vary explicitly. In particular, many measurable features of the same problem instance may be highly correlated, either due to interacting bounds or as a result of the random generation process. Instances with less-likely feature combinations can be attained through iterative local search which successively modifies an instance until it achieves the desired properties [26]. While such instance space search techniques are generally more computationally intensive than parameterised generators, they do provide a reliable method for producing instances with specific target characteristics.

The most common search techniques in use for this application are evolutionary algorithms. Chakraborty and Choudhury [5] and Cotta and Moscato [6] applied this approach to perform statistical average- and worst- case analysis of algorithm performance. More recent work has focused on improving the spectrum of instance hardness [29] and diversity of measured features [9,26]. The success of such work in combinatorial optimisation opens up questions on the use of similar approaches for LP and MIP, adopting a wider range of search algorithms for obtaining difficult-to-design instances, and considering how to best construct the search space for efficient performance.

This paper focuses on developing new instance generation techniques for LP test instances with controllable properties. We present a comparison of a ‘naïve’ random generator with a highly parameterised generator, showing which feature values can be effectively controlled by each method. We also investigate iterative search approaches to find instances which are difficult to design or rarely produced by the generator. These approaches allow practitioners to explore areas of interest in the space of linear programming problems, such as where phase transitions occur or where challenging instances have previously been found. This would not be possible using static tests sets or naïve random generation methods, which provide limited feature control.

Linear programming is considered to be a well-solved optimisation problem. However, solving LP relaxations has the potential to become a bottleneck for solving MIP problems in future as the size of models increases [3]. Indeed, there are examples in the MIPLIB2010 test set where solving the root node relaxation is already a prohibitively

expensive step [20]. However, it is not only the size of instances which makes them challenging to solve. By focusing on variation of the features of smaller instances in this work, we hope to produce a dataset which provides a useful characterisation of LP algorithm performance in terms of variations in problem structure.

The remainder of the paper is organised as follows. In Sect. 2 we describe the characteristics of LP instances that we aim to control. The implementation of a generator for feasible bounded linear programs is described in Sect. 3. Results are presented in Sect. 4 and opportunities for application of the methodology used here to other problems are outlined in Sect. 5 before conclusions are drawn in Sect. 6.

2 Linear programming instances

This section introduces notation, properties and measured features for LP instances. The LP generator produces instances in canonical form:

$$(P) \quad \begin{aligned} \max \ & c^T x \\ \text{s.t. } & Ax \leq b \\ & x \geq 0 \end{aligned}$$

where $A \in \boldsymbol{Q}^{m \times n}$, $b \in \boldsymbol{Q}^{m}$ and $c \in \boldsymbol{Q}^{n}$. The dual program of (P) is given by:

$$(D) \quad \begin{aligned} \min \ & b^T y \\ \text{s.t. } & A^T y \geq c \\ & y \geq 0. \end{aligned}$$

In addition to the primal variables x and dual variables y, we will use the constraint slack variables s and r for (P) and (D), respectively. The instance data is the tuple (A, b, c), which fully describes the canonical form problem.

The matrix form representations (P) and (D) are the forms used in the design of the generator. For feature calculation, we also consider instances as represented by the variable-constraint graph (VC). The variable-constraint graph is a bipartite graph where the disjoint sets of nodes $\{u_i\}$ and $\{v_j\}$ respectively represent variables and constraints in the linear program. An edge exists between nodes u_i and v_j only if the corresponding entry a_{ji} is non-zero. The degree sequences of variable and constraint nodes in this bipartite graph are therefore defined as the number of non-zeros in columns and rows respectively. We use $d(u_i)$ and $d(v_j)$ to denote degree of variable and constraint nodes, respectively.

Table 1 gives the features we consider for measuring properties of LP instances. Features used in this work have been previously used for performance prediction in MIP [17], combinatorial auctions [21] and set covering [18]. This feature set includes properties based on the optimal solution to the LP, which are not suitable for predicting the performance of LP algorithms. However, it is still advantageous to consider these properties since they may affect the structure and difficulty of generated instances.

Statistical features of the constraint matrix, right-hand side and objective coefficients are uniquely defined. However, if the problem has alternative optimal solutions,

Table 1 Features of linear programming instances

Solution features
Number of binding constraints at the optimal point
Number of fractional primal variables at the optimal point
Integer slack vector; the minimum Manhattan distance of the optimal point to an integral point (ignoring feasibility)
Variable constraint graph features
Degree sequence of variable nodes in VC (min/mean/max)
Degree sequence of constraint nodes in VC (min/mean/max)
Coefficient value features
Coefficient statistics $\{a_{ji} \mid a_{ji} \neq 0\}$ (min/mean/max)
Right-hand side statistics $\{b_j\}$
Objective coefficient statistics $\{c_i\}$
Row-normalised coefficient statistics $\left\{\frac{a_{ji}}{b_j} \mid b_j \neq 0\right\}$ (min/mean/max)
Column-normalised coefficient statistics $\left\{\frac{a_{ji}}{c_i} \mid c_i \neq 0\right\}$ (min/mean/max)
Constraint degree-normalised RHS statistics $\left\{\frac{b_j}{d(v_j)}\right\}$ (min/mean/max)
Variable degree-normalised objective statistics $\left\{\frac{c_i}{d(u_i)}\right\}$ (min/mean/max)

the solution features given in Table 1 may not be unique. This phenomenon is further explored when the instance construction method is defined (Sect. 3), and in the results where the generated feature distributions are analysed (Sect. 4.1).

3 Instance generation algorithms

This section describes the implementation of a generator for feasible bounded linear programs. The set of all feasible bounded LPs is termed $\mathcal{P}$, as a subset of the set of all possible LP instances Π. We require that our generator is *correct*, in that for any input parameters a feasible bounded linear program is produced, and *complete*, in that it is capable of producing any instance of the target set. This is in contrast to a *naïve* generator which produces linear programs at random without guaranteeing that they are feasible and bounded.

This problem is approached by breaking the instance generation task into two steps: generation and construction. We introduce an encoding E which admits only feasible bounded instances. The constructor is a deterministic algorithm which converts an encoded instance into a linear program such that it is guaranteed to be feasible and bounded. This algorithm establishes a correspondence between the encoded space and the space of all feasible bounded linear programs. The generator produces random encoded forms, such that it is complete and correct for the encoded space. It is significantly easier to guarantee the properties of completeness and correctness by introducing the intermediate encoding.

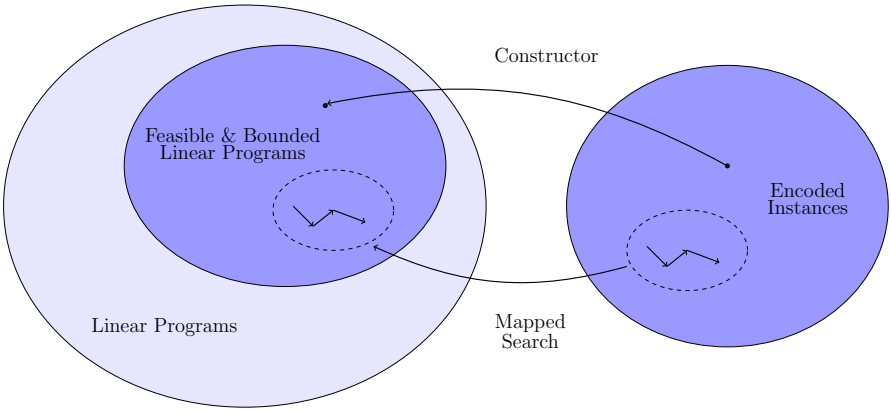

Fig. 1 The constructor maps the instance generation and search problem to a different space by means of an encoded representation of instance data

Completeness does not guarantee that all instances in the target space are equally likely to be generated. We may still need to employ search algorithms to find instances with target properties that cannot be controlled explicitly. The introduction of the constructor simplifies the implementation of search operators by redefining them in the encoded space. The need for repair heuristics to maintain feasibility and boundedness of candidates in the search is thereby avoided and we can guarantee completeness and correctness of the search operators.

Figure 1 summarises these generation and search procedures. The constructor is defined as a mapping from encoded space E such that the image of E is the target space $\mathcal{P}$. Instance generation and local search can then proceed without restrictions in this encoded space.

In Sect. 3.1 we define a naïve generator for general LP instances, which does not guarantee that instances are feasible and bounded. In Sect. 3.2 we define an encoding which admits only feasible bounded LP instances, and a constructor to build instances from the encoded data. We show that this encoding is correct and complete. In Sect. 3.3 we define a generator for encoded forms $e \in E$ of feasible bounded linear programs, and show that it is correct and complete. Section 3.4 gives neighbourhood operators which are developed using the same random processes as each generator.

For both the naïve and controllable generators, input parameters are given which control desired features within the randomisation scheme. After specifying the problem size, the remaining parameters are size-independent, intended to capture instance structure. The key difference between the two generators is that the controllable generator is guaranteed to produce instances which lie in $\mathcal{P}$. Furthermore, the naïve generator cannot be used to search the space $\mathcal{P}$ without the use of inefficient repair operations.

3.1 Naïve generator

The naïve generator G_{Π} for linear programs generates the typical (A, b, c) representation directly. Constraint right-hand side values b and objective function coefficients

c are drawn from normal distributions. For a given instance, the mean and standard deviation of these distributions is chosen randomly. Values c_i and b_j are then drawn from the corresponding distributions. This approach varies the corresponding statistical features directly.

The generator varies features of the constraint matrix A by varying the variable and constraint degree sequences and distribution of coefficients. Considering the sparsity pattern of A to be represented by the variable-constraint graph VC, a degree sequence for both the variable and constraint nodes is first generated. Each degree sequence must sum to the same total number of edges (determined by density) in order to be valid. Algorithm 1 constructs degree sequences by increasing the degree of one variable and one constraint node at each step. The next node is chosen based on the current degree sequence, where weighting parameters (p_v, p_c) alter the choice from completely random (producing uniform degree) to highest degree priority (producing maximum degree of one node before moving on to others). Once the input degree sequences are constructed, edges are assigned with probability given by their respective end nodes to achieve the required sequences.

Edges are added indirectly via this method because the arbitrary sequences may not be satisfiable, and it is computationally costly to find a graph with these exact sequences. Specifically, for bipartite graphs, the sum of constraint degree and sum of variable degree must be equal. We aim to control the distribution of both degree sequences independently via parameters, and it is easier to do so approximately by determining edge probabilities instead of matching an exact sequence. This method is suitable for our purposes as it varies the required statistical features of variables and constraint degree and is relatively efficient. Any unconnected nodes remaining after this process are connected to prevent the algorithm from producing empty constraints or unbounded variables. Non-zero values in the constraint matrix are then drawn from a normal distribution.

Algorithm 1 Constraint generator.

Input: $n \in [1, \infty]$, $m \in [1, \infty]$, $\rho \in (0, 1]$, $\hat{x}$, $\hat{y}$, $p_v \in [0, 1]$, $p_c \in [0, 1]$, $\mu_A \in (-\infty, \infty)$ $\sigma_A \in (0, \infty)$
Output: Constraint matrix $A \in \boldsymbol{Q}^{m \times n}$.

Set target variable degree $d(u_i) = 1$ for randomly selected i, 0 for all others
Set target constraint degree $d(v_j) = 1$ for randomly selected j, 0 for all others
$e \leftarrow 1$
while $e < \rho mn$ **do**
  $s \leftarrow$ draw n values from $U(0, 1)$
  $t \leftarrow$ draw m values from $U(0, 1)$
  Increment the degree of variable node i with maximum $p_v \frac{d(u_i)}{e} + s_i$
  Increment the degree of constraint node j with maximum $p_c \frac{d(v_j)}{e} + t_j$
  $e \leftarrow e + 1$
end while
for $i = 1, \ldots, n$ **do**
  for $j = 1, \ldots, m$ **do**
    $r \leftarrow$ draw from $U(0, 1)$
    if $r < \frac{d(u_i)d(v_j)}{e}$ **then**
      Add edge (i, j) to VC
    end if
  end for
end for
while $\min(d(u_i), d(v_j)) = 0$ **do**
  Choose i from $\{i \mid d(u_i) = 0\}$, or randomly if all $d(u_i) > 0$
  Choose j from $\{i \mid d(v_j) = 0\}$, or randomly if all $d(v_j) > 0$
  Add edge (i, j) to VC
end while
for $(i, j) \in E(VC)$ **do**
  $a_{ji} := N(\mu_A, \sigma_A)$
end for
return A

3.2 Encoding and construction

The naïve generation routine described in Sect. 3.1 is clearly not sufficient to define a controllable generator $G_{\mathcal{P}}$ which is complete and correct. Instead we define an encoding for LP instances which stores the constraint matrix A along with a complete optimal solution $(\hat{x}, \hat{y}, \hat{r}, \hat{s})$. The constructor produces a feasible and bounded linear program from this encoded form.

To design an encoding for this problem, we consider the key characteristic of instances in $\mathcal{P}$. The primal problem (P) and dual problem (D) must both be feasible. Therefore, if the encoding stores at least one solution to each problem, it can be guaranteed to represent a feasible bounded linear program. We take this one step further by storing optimal solutions to the primal and dual which satisfy the complementary slackness conditions. This reduces the number of possible unique encoded forms for each instance, avoiding one source of redundancy in the encoding.

The encoding stores the constraint matrix A given for the canonical form of the primal problem. An optimal solution is encoded using a binary vector β and a non-negative vector α. The constructor assigns values from α to either primal or dual variables according to the binary value in β, ensuring the primal and dual solutions are

complementary. Ensuring feasibility of the given solution is then the only requirement to design an instance for which these solutions are optimal.

The domain of the encoding E is all tuples (n, m, A, α, β) which satisfy

$$\begin{array}{ll} n, m \in \boldsymbol{N} & \\ A \in \boldsymbol{Q}^{m \times n} & \\ \alpha \in \boldsymbol{Q}^{m+n} & \alpha_i \geq 0 \\ \beta \in \{0,1\}^{m+n} & \sum \beta_i = m. \end{array} \tag{1}$$

Given an input in E, the constructor must return a feasible bounded linear program with n variables and m constraints in canonical form (A, b, c). The constructor is defined in Algorithm 2.

Algorithm 2 Constructor for feasible bounded linear programs.

Input: Encoded form $(n, m, A, \alpha, \beta) \in E$
Output: Linear program in canonical form (A, b, c)
1: **for** $i = 1, \ldots, n$ **do**
2: $\quad \hat{x}_i \leftarrow \beta_i \alpha_i$
3: $\quad \hat{r}_i \leftarrow (1 - \beta_i)\alpha_i$
4: **end for**
5: **for** $j = 1, \ldots, m$ **do**
6: $\quad \hat{y}_j \leftarrow (1 - \beta_{j+n})\alpha_{j+n}$
7: $\quad \hat{s}_j \leftarrow \beta_{j+n}\alpha_{j+n}$
8: **end for**
9: $b \leftarrow A\hat{x} + I_m \hat{s}$
10: $c \leftarrow A^T \hat{y} - I_n \hat{r}$
11: **return** (A, b, c)

Proposition 1 *Given any encoded form $e \in E$, the constructor produces a feasible bounded linear program. Moreover, the vectors $\hat{x}$, $\hat{s}$, $\hat{y}$ and $\hat{r}$ provide optimal solutions to the LP and its dual problem.*

Proof Let vectors $\hat{x}$, $\hat{s}$, $\hat{y}$ and $\hat{r}$ be constructed as in Algorithm 2 for a given encoded form $e \in E$. Due to the definitions of b and c in this algorithm, and non-negativity of α as required by the encoding, $\hat{x}$ and $\hat{y}$ are feasible solutions to the linear programs (P) and (D), respectively. It follows from weak duality that both problems are also bounded.

Furthermore, $(\hat{x}, \hat{s})$ and $(\hat{y}, \hat{r})$ satisfy the complementary slackness conditions, since

$$\hat{x}_i \hat{r}_i = \beta_i (1 - \beta_i)\alpha_i^2 = 0, \quad \forall i = 1, \ldots, n$$

and

$$\hat{s}_j \hat{y}_j = \beta_{j+n}(1 - \beta_{j+n})\alpha_{j+n}^2 = 0, \quad \forall j = 1, \ldots, m.$$

By the complementary slackness theorem, $\hat{x}$ and $\hat{y}$ are optimal solutions to (P) and (D) respectively. □

Proposition 2 *Any feasible bounded linear program has at least one encoded form* $e \in E$.

Proof Consider any feasible and bounded linear program (P). Without loss of generality, we may assume that (P) is given in canonical form and defined by (A, b, c). Then, the dual linear program (D) of (P) must also be feasible and bounded. Furthermore, linear programs (P) and (D) in standard forms must have basic optimal solutions. Let $(\bar{x}, \bar{s})$ and $(\bar{y}, \bar{r})$ be basic optimal solutions of (P) and (D), respectively, where $\bar{s}$ and $\bar{r}$ are the corresponding slack values of the constraints. Let us consider vectors $p = (\bar{x}\ \bar{s})$ and $d = (\bar{r}\ \bar{y})$. The vectors p and d must be non-negative and have at most m and n positive entries, respectively. Moreover,

$$p_i d_i = 0, \quad \forall i \in \{1, \ldots, n+m\}.$$

Algorithm 3 Construction of encoded form.

```
S ← ∅
for i = 1, ..., m + n do
  if d_i > 0 then
    β_i ← 0 ; α_i ← d_i
  else
    if p_i > 0 then
      β_i ← 1; α_i ← p_i
    else
      β_i ← 0; α_i ← 0
      Add i to the set S
    end if
  end if
end for
```

Consider vectors α and β constructed by Algorithm 3. Then, by our construction, we have

$$\sum_i \beta_i + |S| \geq m$$

since d has at most n positive entries and

$$\sum_i \beta_i \leq m$$

since p has at most m positive entries. Moreover, for any $i \in S$ we have $d_i = p_i = 0$. In order to satisfy the encoding requirement that $\sum_i \beta_i = m$ we take any subset $S' \subset S$ with $|S'| = m - \sum_{i=1}^{n+m} \beta_i$ and set $\beta_i = 1$ for all $i \in S'$.

To complete the proof we show that (n, m, A, α, β) is an encoded form of (P). Let $(\hat{x}, \hat{s})$ and $(\hat{y}, \hat{r})$ be vectors constructed in Algorithm 2 for (n, m, A, α, β). Then, by our construction of α and β, we have

$$(\hat{x}, \hat{s}) = (\bar{x}, \bar{s}) \quad \text{and} \quad (\hat{y}, \hat{r}) = (\bar{y}, \bar{r}).$$

Consequently, Algorithm 2 returns the same right-hand side and objective coefficients b and c. □

Proposition 1 implies that the encoding is correct, since Algorithm 2 will produce a feasible bounded linear program given a valid encoded form. Proposition 2 implies that the encoding is complete since for any feasible bounded linear program there exists at least one corresponding encoded form. This establishes a correspondence between the encoded space and target problem space, meaning we now only need to define a complete and correct generator G_E for encoded forms. Clearly, it is significantly easier to develop a generator for encoded forms satisfying (1) than to attempt to extend the naïve generator to produce only feasible bounded instances.

The constructor is guaranteed to produce a unique output instance for a given encoded input. However, the mapping from encoded forms is not one-to-one with respect to the set of feasible bounded LPs encoded in the form (A, b, c). Algorithm 3, which constructs an encoded form from a feasible bounded LP, shows that this redundancy can occur where multiple optimal solutions exist for a given linear program. An algorithm which generates random encoded forms will favour instances with multiple representations after construction, so instances with multiple solutions may be over-represented in a distribution of feasible bounded linear programs using this method.

3.3 Controllable generator

The generator G_E produces encoded forms $(n, m, A, \alpha, \beta) \in E$. Since the constraint matrix A is passed unchanged through the constructor, G_E uses Algorithm 1 for its first stage. As described in Sect. 3.1, this process controls features of the constraint matrix.

The second stage generates the optimal solution for the constructed instance, including optimal values of the primal and dual variables, and slack values for all primal and dual constraints at the optimal point. The generator produces vectors α and β which define a complete primal-dual solution in the form required by the constructor. The number of primal and slack basic variables, number of binding constraints, number of fractional primal solution values and degree of fractionality of the optimal solution are controlled by input parameters.

Algorithm 4 constructs a basic variable set for the optimal solution containing the required number of primal and slack variables. The number of primal and slack variables is controlled by the basis ratio parameter γ. Slack variable values are chosen from a parameterised log-normal distribution, hence they are non-negative. Primal variable values are chosen from a parameterised log-normal distribution and rounded to the nearest integer. A subset of primal variables are chosen to take on fractional values at the optimum. For each of these, a value in $[0, 1]$ is subtracted from the integral value. These fractional components are drawn from a parameterised symmetric beta distribution to control the distance from the optimal point to its simply rounded point.

Algorithm 4 Solution generator.

Input: Generator parameters $n \in [1, \infty]$, $m \in [1, \infty]$, $\gamma \in [0, 1]$, $\lambda \in [0, 1]$, $a \in (0, \infty)$, $\mu_p \in (-\infty, \infty)$, $\sigma_p \in (0, \infty)$, $\mu_s \in (-\infty, \infty)$, $\sigma_s \in (0, \infty)$
Output: Encoded solution (α, β)
$k \leftarrow [\gamma \min(n, m)]$
$P_{opt} \leftarrow$ Choose k indices from $\{1, \ldots, n\}$ without replacement.
$S_{opt} \leftarrow$ Choose $m - k$ indices from $\{1, \ldots, m\}$ without replacement.
$P_{frac} \leftarrow$ Choose $[\lambda n]$ indices from $\{1, \ldots, n\}$ without replacement.
for $i = 1, \ldots, n$ **do**
  $X_1 \leftarrow$ draw from LogNormal(μ_s, σ_s)
  $X_2 \leftarrow$ draw from Beta(a, a),
  if $i \in P_{opt}$ **then** $\beta_i := 1$ **else** $\beta_i := 0$ **end if**
  if $i \in P_{frac}$ **then** $\alpha_i := \lceil X_1 \rceil - X_2$ **else** $\alpha_i := \lceil X_1 \rceil$ **end if**
end for
for $j = 1, \ldots, m$ **do**
  $X_3 \leftarrow$ draw from LogNormal(μ_s, σ_s).
  if $i \in S_{opt}$ **then** $\beta_{n+j} := 1$ **else** $\beta_{n+j} := 0$ **end if**
  $\alpha_{n+j} := X_3$
end for
return (α, β)

Given the choice of primal and slack variable indices in Algorithm 4, where $k = [\gamma \min(n, m)]$, we have $0 \leq k \leq n$ and $0 \leq m - k \leq m$ as $0 \leq k \leq m$, so choice without replacement is valid. Note that k is produced by rounding to the nearest integer, so that varying the scale-independent parameter γ in the range $[0, 1]$ can produce any integer value of k on the interval $[0, \min(n, m)]$. Furthermore, given the binary choice of values in β, $\beta \in \boldsymbol{B}^{n+m}$ and $\sum_{i=1}^{n+m} \beta_i = \left|P_{opt}\right| + \left|S_{opt}\right| = k + (m - k) = m$ as required by Eq. (1).

Elements of $\{\alpha_i \mid i \leq n\}$ are first drawn from $\lceil X_1 \rceil \in [1, \infty)$, since the log-normal distribution produces only positive numbers and these values are rounded up. For all $\{\alpha_i \mid i \in P_{frac}\}$, values drawn from $X_2 \in [0, 1]$ are subtracted, which maintains $\alpha_i \geq 0$. Elements of $\{\alpha_i \mid i > n\}$ are drawn from $X_3 \in (0, \infty)$, which also gives $\alpha_i > 0$. Therefore $\alpha \geq 0$ as required by Eq. (1). Solution pairs (α, β) produced by the generator are therefore valid encoded forms. Including the constraint matrix generated by Algorithm 1, gives a string $(A, \alpha, \beta) \in E$, so the generator algorithm is correct.

As above, $\left|P_{opt}\right| = [\gamma \min(n, m)]$ and $\left|S_{opt}\right| = m - [\gamma \min(n, m)]$. All non-basic primal variables are zero, so fractional primal (optimal) variables are the subset of basic variables for which α_i is fractional. This is controlled by the parameter λ, so $\left|P_{frac}\right| = [\lambda \left|P_{opt}\right|]$.

Fractional components for each fractional primal variable are drawn from a symmetric beta distribution. The beta distribution is ideal for our purposes as it is distributed on $(0, 1)$. Once an integral value is chosen for the solution variables, the parameters p and q of this distribution control the distance to the nearest integer point. We use a symmetric distribution $(a := p = q)$ so that values are equally likely to occur in $(0, 0.5)$ and $(0.5, 1)$. The combined parameter a must be non-negative. The resulting symmetric beta distribution is uniform for $a = 1$, centrally skewed for $a > 1$, and edge-skewed for $a < 1$.

This gives the algorithm control over the total fractionality feature: central skew results in fractional primal values which maximise this feature; edge skew minimises this feature. As such the degree of rounding required from the optimal point to the nearest integer solution is controlled by this skew parameter.

Fractional values are subtracted from the base integer values, so calculating the average fractional component requires splitting the distribution in half (since F is the Manhattan distance to the simply rounded point). The distribution is symmetric about 0.5, so

$$E\left[|\alpha_i - [\alpha_i]|\right] = E\left[x \sim Beta(a,a) \mid x < 0.5\right]$$

Using the probability density function of the beta distribution, $f(x; p, q)$, and the definition of the partial beta function, the mean fractionality as a function of the parameter a for the symmetric beta distribution is calculated using the partial integral:

$$\begin{aligned} E\left[|\alpha_i - [\alpha_i]|\right] &= \frac{1}{B(p,q)} \frac{1}{0.5} \int_0^{0.5} x\, f(x; p, q)\, dx \\ &= \frac{2}{B(a,a)} \int_0^{0.5} x^a (1-x)^{a-1}\, dx \\ &= 2\, \frac{B(0.5; a+1, a)}{B(a,a)} \end{aligned}$$

Finally, we can calculate the expected value of total fractionality F as:

$$\begin{aligned} E[F] &= \left|P_{frac}\right| \; E\left[|\alpha_i - [\alpha_i]|\right] \\ &= 2[\lambda\gamma \min(n,m)] \frac{B(0.5; a+1, a)}{B(a,a)} \end{aligned}$$

where $B(p,q)$ is the beta function and $B(x; p, q)$ is the incomplete beta function.

With each of these features altered by the generator input parameters, the generator has the capacity to produce a given distribution of features by varying parameter values appropriately in the generation scheme. The relations derived in this section are useful for this purpose. For example, density and basis ratio are linearly dependent on relevant generator parameters. Therefore a uniform generator would produce instances by selecting those parameters from uniformly independent distributions in the given ranges. The number of fractional variables is the combined result of basis ratio and integrality violations, so has joint quadratic dependence on input parameters.

The mean fractionality feature has more complex relationship to input parameters. Generating instances with a maximum value of fractionality (where all fractional components are close to 0.5) requires very large values of the parameter α. The log-normal distribution was therefore selected for this parameter to ensure large values were produced with high enough probability. This choice leads to approximately uniform variation in the mean fractionality feature.

The distributions chosen in the generator algorithm are selected in order to produce a uniform spread of feature values across the bounds of the feature space. This may

be a useful distribution to choose in a feature-performance learning application, since it is advantageous to vary characteristics of the instance data independently. However, depending on the application, a different choice of distribution may be appropriate. We choose the uniform distribution here as an example to demonstrate the analysis required to verify the expected feature output distributions.

3.4 Instance space search operators

The generators defined above can be used to define neighbourhood operators which replace part of the instance data. Consider the naïve generator G_{Π}. Given an instance with m rows and n columns, generate a new instance using G_{Π} with one row and n columns. The generated data replaces a single row of the constraint matrix A, and the corresponding value b_j from the right-hand side. A second neighbourhood operator can be defined which generates a new instance with m rows and one column, replacing a column of A and an objective coefficient c_i.

Note that starting from a feasible bounded instance, these naïve operators may generate a neighbour which is infeasible or unbounded. Local search in the target instance set $\mathcal{P}$ will therefore require neighbours to be rejected at some steps, which impacts search progress, or a repair heuristic to be applied, which may introduce bias.

Alternatively, neighbourhood operators can be defined in the encoding space. Given an encoded form with m rows and n columns, generate a new instance using $G_{\mathcal{P}}$ with one row and n columns. This generates a new row of A as above, and corresponding values α_i and β_i. A new encoded form is produced by replacing a randomly chosen row of A and the corresponding elements of α and β with the new values. A second operator can be defined by replacing a single column of A and corresponding encoded solution values. The constructor builds a new instance from the encoded form which is guaranteed to be feasible and bounded. This neighbourhood operator is therefore correct with respect to $\mathcal{P}$.

To verify that neighbourhood operators in the encoded space are also complete, we observe that the generator G_E is complete with respect to the encoding E. Any two encoded forms $e, e' \in E$ can both be produced by G_E, therefore all of their rows can be produced by the neighbourhood operator. We can therefore produce e' from e by a finite number of applications of the neighbourhood operator. Since the constructor maps E to $\mathcal{P}$ this operator connects the search space and is therefore complete.

4 Experimental results

This section compares the diversity of instances produced by the naïve and controllable generators in feature and performance space. The naïve generator, as expected, produced many infeasible or unbounded instances, while the controllable generator only produced feasible and bounded instances. However, the diversity of the feasible and bounded instances produced by the controllable generator, with its randomised parameters, could be improved since there are still gaps in the feature space where new instances could potentially be generated. We apply search in feature space to fill

Table 2 Generator parameters used in experiment

Parameter	Value/distribution
Variables n	50
Constraints m	50
Density ρ	$U(0.1, 1)$
High degree variables p_v	$U(0, 1)$
High degree constraints p_c	$U(0, 1)$
Coefficient Mean μ_A	$U(-2, 2)$
Coefficient StDev σ_A	$U(0.1, 10)$
Primal versus slack basis γ	$U(0, 1)$
Fractional primals λ	$U(0.1, 1.0)$
Beta fractionality a	$LN(-0.2, 1.8)$

these gaps. Additionally, we use search in performance space to produce instances which are harder for several LP algorithms.

Generation and search is implemented in Python 3.6.3, with a C++ extension using the Coin LP callable library to facilitate feature computation and generation of MPS files to be read by solvers. The pseudo-random number generator included with the NumPy library (version 1.14.2) for Python is used for all random number generation. Generation, feature search and performance search results are reproducible for a given 32 bit seed value for the NumPy random generator. Coin LP 1.16.11 is used to test primal simplex, dual simplex and barrier solvers on generated instances. The CLP algorithms are run using their respective default settings (for simplex, CLP uses steepest edge as its pivot rule). Computations are run on an Ubuntu 17.10 virtual machine with 16 virtual cores and 60GB virtual memory. The code required to reproduce the experiments in this section is available via the Zenodo repository [4].

4.1 Parameterised generation

The full feature set presented in Table 1 contains some properties of LPs that can be controlled directly in the generation process, and others that will need to be addressed through search mechanisms. We demonstrate the application of the generator by producing a diverse set of instances where feature distributions are independently varied across their maximum ranges. In particular, we vary the number of binding constraints, number of primal variables with fractional values at the optimal point, total fractionality (Manhattan distance of the integer slack vector), constraint coefficient density and variable/constraint degree statistics. We investigate fixed size instances of 50 variables and 50 constraints. The remaining generator parameters are independent of instance size, and the distributions used for each one are given in Table 2. These parameter ranges are chosen to demonstrate the relative feature variation which can be achieved using these algorithms.

To generate each instance, parameter values are sampled from the distributions given in Table 2. These parameters are used to run the generator algorithm. The generated data set consists of 1000 instances from the controllable generator and 3000

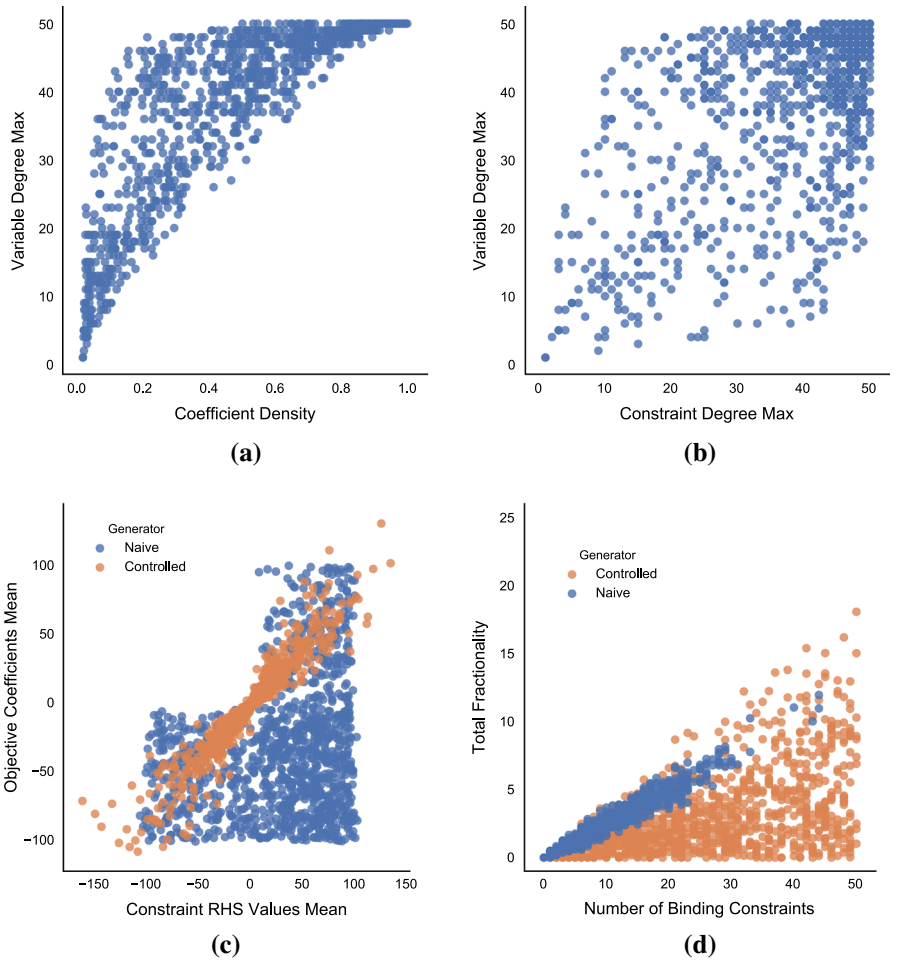

Fig. 2 Distribution of instances in feature space; **a**, **b** show features of the constraint matrix, which are common to the two generators, **c**, **d** compare distributions of features where the generators differ, showing only feasible bounded instances

instances from the naïve generator. The larger sample size is required for the naïve generator to produce as many feasible bounded instances as the controllable generator since only 37.5% of instances generated using the naïve method satisfy this property.

Figure 2a, b show features of the constraint matrix, where each scatter plot point represents the feature values of a generated instance. The naïve and controllable generators use the same method to generate the left-hand sides of constraints, therefore they produce identical distributions in this feature space. The generator for A constructs degree sequences for the variable-constraint graph. This allows the variable and constraint degree statistics to be varied independently of one another. In contrast to a uniform random approach, which would produce approximately uniform degree in most cases, the generator is able to improve feature diversity in this space.

Figure 2c shows the joint distribution of mean values of the objective coefficients c_i and constraint upper bounds b_j. This figure shows only feasible bounded instances. The naïve generator achieves more diversity in this feature space; it produces instances in the second quadrant while the controllable generator is more restrictive. The controllable generator can also be used to produce feasible bounded instances in the second quadrant, however, examination of the constructor algorithm shows that such instances are not likely to be useful in algorithm testing. Recalling that the constructor algorithm calculates c_i and b_j from the encoded form as

$$c_i = A_i^T y - r_i$$
$$b_j = A^j x + s_j$$

where $r_i, s_j \geq 0$, it is easy to see that we could drive instances towards the lower right corner of Figure 2c by simply choosing large values for the reduced cost and slack variables at the optimal point. However, this would clearly produce uninteresting instances, where many of the primal and dual constraints are inactive near the optimal point. Constraints with such large slack values may be entirely redundant in the formulation, resulting in an easy to solve instance.

Figure 2d shows the joint distribution of the number of binding constraints at the optimal point and total fractionality of the primal solution values. This demonstrates the key additional parameters available to the controllable generator; it can explicitly set the primal, dual, slack and reduced cost values. By contrast, the naïve generator occupies a very narrow band in this feature space. More importantly, the parameters of this generator have no bearing on these features, so it is not possible to alter the input parameter distributions in order to achieve more variation.

The controllable generator does not produce a uniform distribution across the space. Our input parameters uniformly vary the number of constraints which are binding at the optimal point. The number of binding constraints places an upper limit on the number of non-zero primal variables at the optimal point. Therefore, if there are k binding constraints, the maximum value for total fractionality is $0.5k$. As a result this two dimensional distribution is not jointly uniform. The input parameters could be altered to populate the upper right corner if required, by increasing the parameters controlling solution fractionality and number of binding constraints at the optimal point.

The effects of solution degeneracy result in a mismatch between our expected feature distribution (which we can calculate using formulae derived in Sect. 3.3) and the actual distribution. While the constructor guarantees (by Proposition 1) that the solution specified in the encoded form is optimal, it does not guarantee that it is unique. Therefore when instance features are calculated by solving the LP from scratch, a different optimal point may be used, and the measured feature value may differ from the design value.

This degeneracy effect is further explored in Fig. 3, which shows the likelihood that the solution found using simplex differs from the constructed optimal solution. An instance produced by the constructor will have degenerate optimal solutions if the selected primal variables in the design solution $\hat{x}$ correspond to a singular submatrix of the constraint matrix A. Overall this occurs 32.5% of the time in instances produced by

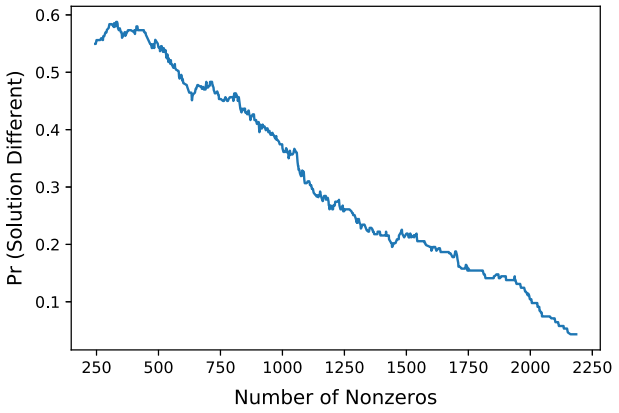

Fig. 3 Proportion of instances with the given number of non-zero entries in the constraint matrix for which the solution found by the simplex algorithm differs from the designed optimal solution

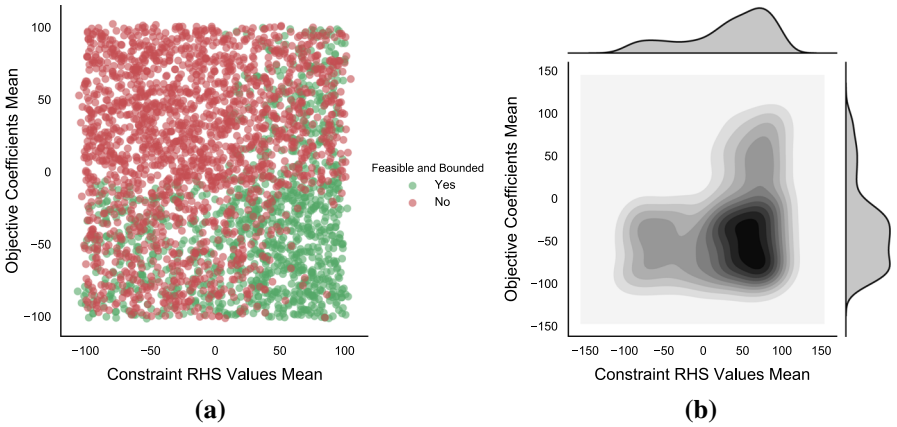

Fig. 4 Observed phase transition in feasibility related to the distribution of objective coefficient and constraint right-hand side values; **a** shows the uniform sampling of instances across a two dimensional feature space using the naïve generator; **b** shows the estimated probability density for feasible bounded instances in this space

the controllable generator. Figure 3 shows that there is a high likelihood of degeneracy where the constraint matrix is low-density. The probability of degeneracy tends towards zero where all entries in the matrix are non-zero. This is consistent with the likelihood that a matrix with random entries is singular.

Figure 4a shows the original sample points of the naïve generator, with both feasible bounded and infeasible or unbounded instances shown. The set of *all* naïvely generated instances is independently uniformly distributed in this two dimensional space. As previously mentioned, 37.5% of instances from the naïve generator were feasible and bounded, and we can observe a clear relationship between position in this feature space and the likelihood that an instance is feasible and bounded. This is presented in Fig. 4b as a probability density function. This phase transition does not necessarily

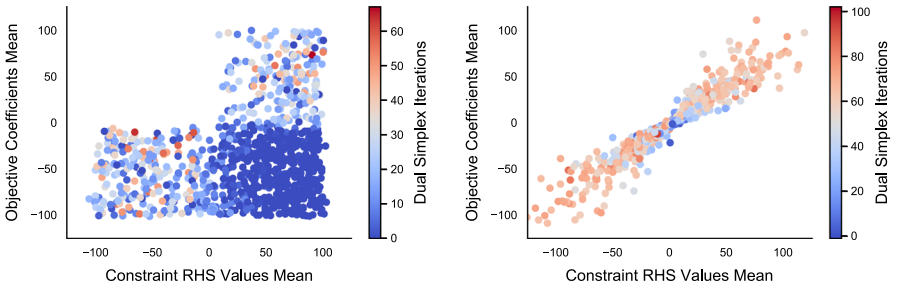

Fig. 5 Variation in number of iterations required for the dual simplex algorithm to solve instances around the phase transition region for the two generators; **a** shows instances produced using the naïve generator, **b** shows instances produced using the controllable generator

apply to all linear programs; it is specific to the parameter distribution and generator algorithm we have applied.

As an intuition for why we observe this phase transition, note that if $b_j > 0$ for all constraint indices j, then the trivial solution ($x = \bar{0}$, the zero vector) is feasible to the primal program (P). Similarly, if $c_i < 0$ for all variable indices i, then the solution $y = \bar{0}$ is feasible to the dual program (D). This scenario becomes more likely as the mean values of these coefficients are increased relative to their standard deviation. Therefore we expect instances in the second quadrant of Fig. 4a to be more likely to be feasible and bounded. However, we further note that such instances are likely to be uninteresting, as the trivial solution is likely to be optimal, resulting in an easily solvable linear program. This is consistent with our previous observations of how to generate instances in this region using the controllable generator.

We are therefore more interested in behaviour near the phase transition, where zero-feasibility is not guaranteed and only certain combinations of generated constraint hyperplanes are likely to give feasible bounded instances. Indeed as we observe in Fig. 5a, instances in the transitional quadrants, where feasibility and boundedness is less predictable, are harder to solve using the dual simplex algorithm. Furthermore, the controllable generator, which produces instances almost exclusively in this region as shown in Fig. 5b, results in harder to solve instances.

The disparity in difficulty between the instances produced by the two generators may also be explained by the ability of the controllable generator to set the number of binding constraints. Figure 6 shows a strong correlation between the number of binding constraints at the optimal point and the number of dual simplex iterations required to solve an instance. The naïve random generator is not able to control this feature, and the resulting instances have less than 60% of constraints active at the optimal point. It is clearly advantageous to be able to control this feature when generating instances to challenge simplex algorithms.

Qualitative assessment of the generated instance data shows that the controllable generator produces greater variation of features of interest (some of which are correlated with difficulty), and as a result, more challenging instances on average. The generated test set still lacks some diversity. In particular there are regions in feature

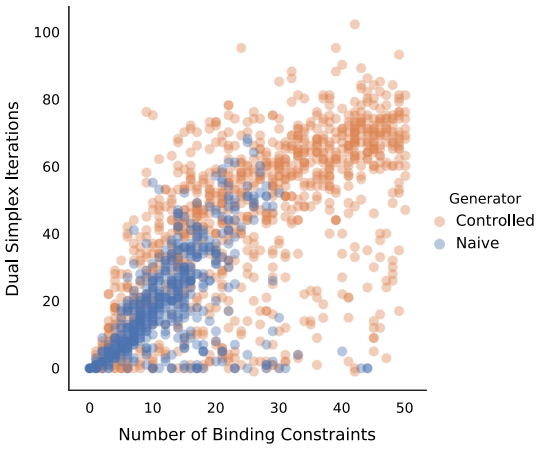

Fig. 6 Comparison of the two generators in terms of instance difficulty for the dual simplex algorithm. The controllable generator produces harder instances because it is able to directly vary the number of binding constraints at the optimal point

space which are missing completely from our test set, and it seems likely that we can find more variation in performance space.

4.2 Local search in instance space

Our search efforts in feature space focus on the missing region in Fig. 2c. The generators used in Sect. 4.1 did not produce feasible bounded instances in the fourth quadrant. A local search algorithm can be applied to generate new instances in this space.

These results compare performance of a local search algorithm using operators based on the naïve and controllable generators, where the objective is to minimise distance to the target point $(-100, 100)$. For each search run, an initial instance is selected by randomly sampling 20 instances from the existing data set and choosing the one which is closest to the target point. This ensures searches are started from varied, but still reasonably 'good', instances. At each step, the algorithm generates a single neighbour, accepting it as the incumbent if it is feasible and bounded and improves on the objective. Each process runs for 10,000 search steps, and we repeat each run 100 times to assess the average rate of improvement.

Figure 7a shows the original data set (from which starting instances are drawn), and sampled points for each search run at step 1000. Figure 7b shows the progression in the objective over the course of the search. Numerical results are given in Table 3. Local search using the controllable neighbourhood operator clearly converges more quickly on the target point than those using the naïve operator. This is likely due to the high rejection rate of neighbours in the naïve process. 40–50% of neighbouring instances produced using the naïve operator are infeasible or unbounded and thus immediately rejected.

Figure 7c, d show that no new instances were found which were harder to solve than those in the original generated data sets. However, for both primal and dual simplex, the number of simplex iterations required on average to solve instances generated by this target-point construction is higher than that of distributions of randomly generated instances. Following from discussion of the feasibility phase transition in Sect. 4.1, it

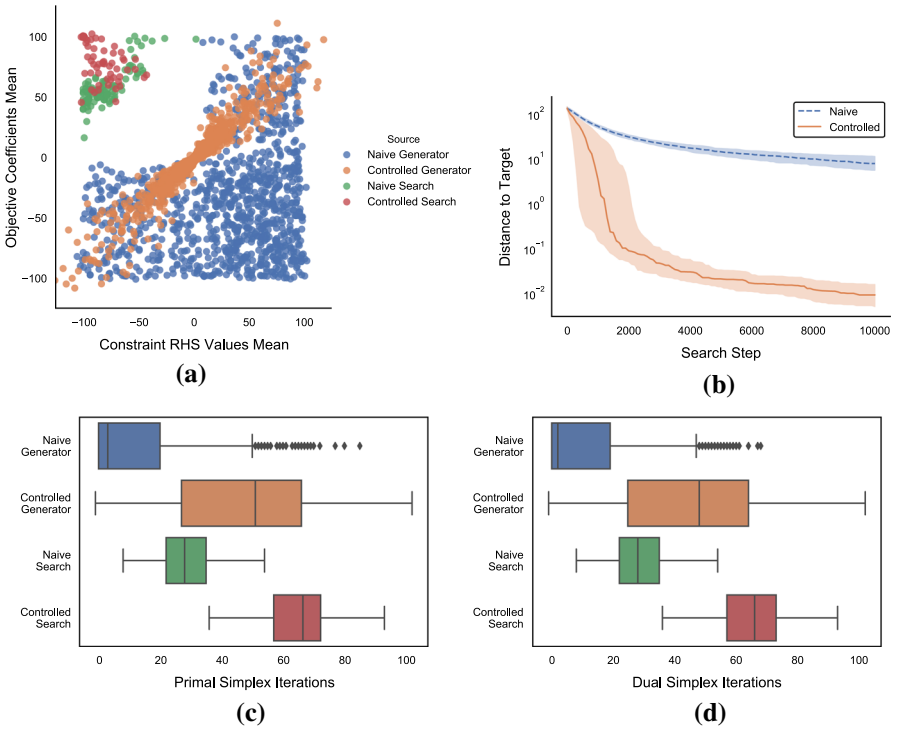

Fig. 7 Local search results in feature space; **a** shows original generated instances and results of local search after 1000 steps, **b** shows progress by instance space search towards the target point in feature space. The central line shows the median distance to target point at the given iteration, while the shaded regions show upper and lower quartiles. Hardness is compared in **c**, **d** for each instance set, showing extreme values and quartile ranges for the naïve and controllable generators and the results of local search

Table 3 Feature space search results

Method	Step	Distance to target point	
		Median	Min
Controllable search	0	129.686	107.561
	100	96.957	0.380
	1000	3.479	0.022
	10,000	0.009	0.000
Naïve search	0	129.476	107.561
	100	118.289	96.888
	1000	52.075	36.016
	10,000	7.688	0.001

Shows the distance to target point achieved by local search at a given step. Results are aggregated over 100 trials for each neighbourhood operator

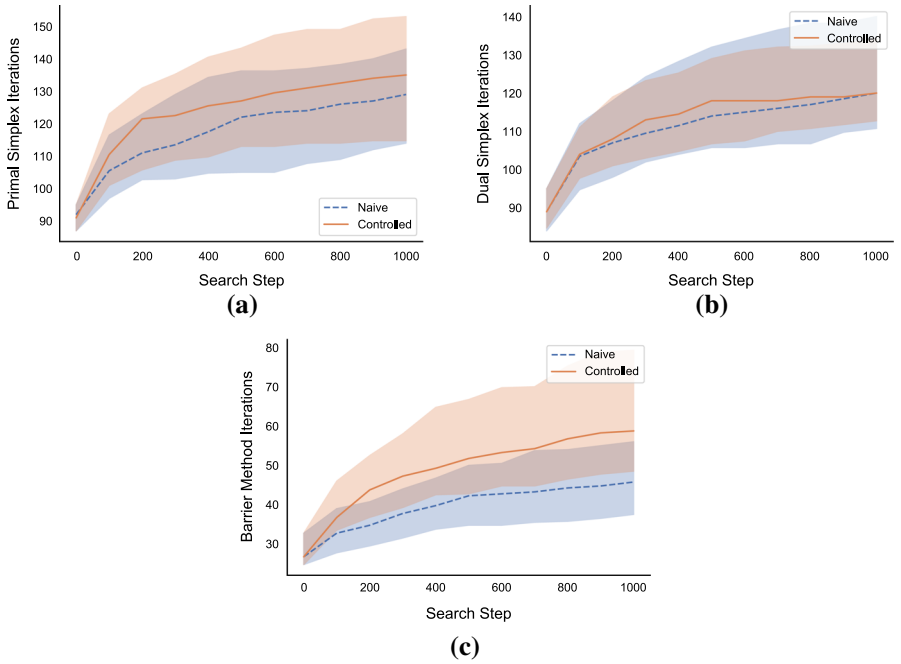

Fig. 8 Progression of local search runs which aim to generate harder instances; **a** shows primal simplex, **b** shows dual simplex, **c** shows barrier algorithm iterations. In each run a local search algorithm attempts to increase the required number of iterations at each step. The central line in each figure represents the median progression of search runs for a given method, and the shaded region indicates the upper and lower quartiles. The darkened region indicates the overlap in interquartile range between methods. Ranges are calculated based on a series of 100 runs with 1000 local search steps for each performance metric

is unlikely that trivial solutions exist for instances near the target point. As a result, focusing on this region of the feature space by using target-point search yields a more challenging distribution of instances.

The presence of a significant number of difficult outliers in the distribution generated by the naïve generator, as shown in Figs. 7c, d, indicates that while hard instances may still be produced by random search, this is not the norm. The instances produced by the controllable generator and target point search processes have more consistent difficulty, while the naïve generator 'stumbles upon' hard instances.

Figure 8 shows search progression using the naïve and controllable operators where the objective is to increase the number of iterations required to solve the instance using primal simplex, dual simplex and barrier algorithms. The algorithm proceeds in the same manner as feature space search, accepting new instances which increase the difficulty metric for the target algorithm. Each run of local search is terminated after 1000 steps. Numerical results are given in Table 4. The controllable search operator in general produces hard instances more effectively than the naïve operator. However, the difference in performance between the two methods is not as pronounced as in feature space search.

Table 4 Performance space search results, showing the number of iterations required by a target algorithm to solve the current LP instance at a given local search step

Method	Step	Primal simplex			Dual simplex			Barrier method		
		Q_1	Q_2	Q_3	Q_1	Q_2	Q_3	Q_1	Q_2	Q_3
Controllable generator	–	27	51	66	24	48	64	13	15	17
Controllable search	0	87	91	95	84	89	95	25	27	33
	200	105	121	131	101	108	119	37	44	52
	500	113	127	143	106	118	129	43	52	67
	1000	114	135	153	112	120	134	48	59	79
Naive generator	–	0	4	20	0	2	19	7	9	12
Naive search	0	87	92	95	84	89	95	25	27	33
	200	102	111	123	98	107	118	29	35	41
	500	105	122	136	105	114	132	35	42	50
	1000	114	129	143	110	120	140	37	46	56

This table gives median, upper and lower quartiles aggregated over 100 search runs for each neighbourhood operator and target algorithm. Statistics for randomly generated instances (without search) are shown for comparison

The results of performance space search demonstrate the benefits of local search in instance space. The initial test set produced by the controllable generator results in, at best, instances which take 102 iterations to solve with the primal simplex algorithm. In general, instances are much easier, with a median value of 51. Local search, which maximises difficulty for the primal simplex algorithm, produces instances which take more iterations to solve, with a median of 135 and maximum of 217. In each case (randomly generated test set and a single search run), a total of 1000 instances must be evaluated using the target algorithm. These procedures therefore require approximately the same amount of computational effort, but local search is able to produce much harder instances. This result demonstrates that a guided generation process can be an effective method to generate instances with characteristics which occur rarely in the generated data sets.

5 Further applications

The task of implementing a generator which is guaranteed to produce feasible bounded linear programs is approached in this work by introducing an alternative encoding for instances. This shifts the task of generating instances to an alternative space which is mapped to the target problem space by a deterministic constructor. The same principle can be applied to other optimisation problems where it is necessary to control a particular decision property of generated instances.

Here we briefly outline a potential approach to defining an encoding for Travelling Salesman Problem (TSP) instances. When generating instances to compare performance of local search improvement heuristics such as those defined by Lin and Kernighan [22], only feasible problems are of interest. For any feasible TSP instance,

the weighted graph defining the problem must have at least one Hamiltonian cycle. An encoding for such a problem could be composed of an ordering of vertices with corresponding weights (defining a known cycle), and a separate weighted undirected graph. The constructor would then take the weighted graph and add weighted cycle edges as defined by the ordering. The resulting instance is guaranteed to have at least one feasible tour, with a known distance.

Random generators and search operators can be easily defined in this encoded space. A generator simply needs to choose a random permutation of the indices $1, \ldots, N$ and generate a random graph on N vertices. Similarly, local search operators could reorder the permutation, add or remove edges from the random graph, or scale a subset of weight values in the encoding. These operators maintain valid encodings, and via the definition of the constructor can be shown to be complete and correct for the space of feasible TSP instances.

The constructor framework can also be extended to generate general mixed integer programs. The LP-specific generators and search methods developed in this paper are easily extended to MIP generators by adding integrality constraints. Generated MIP instances would be guaranteed to have feasible relaxations, thus avoiding trivial cases which can be proven infeasible without testing key components of the MIP solver. Furthermore, control over solution fractionality in the LP generator allows the degree of integrality constraint violation to be set at the root node.

A more general MIP encoding could include valid inequalities based on the constraint set to control the shape of the integer hull. Such an approach could be used to guarantee integer feasibility (or infeasibility) of generated instances and to set the strength of LP bounds.

6 Conclusion

This paper proposes a method to generate linear programming instances with controllable properties. A two stage algorithm is developed which generates instances using an alternative representation and maps them to the original problem space. This process guarantees that the resulting instances are feasible and bounded.

The controllable generator is theoretically capable of producing any feasible bounded LP given the right parameter choices. It enables a diverse range of instances to be produced by explicitly varying selected structural features of LP instances. The feature set used here includes properties of the instance data and of the optimum solution of the LP. Although the solution feature set would not allow for prediction of LP algorithm performance, controlling these properties does allow for control over instance difficulty.

In cases where particular feature values are very rarely produced by the generator, local search methods can be applied to generate such instances by iteratively converging on the required values. These search methods are shown to be more effective in the encoded space than the original problem space. Local search methods can also be used to generate instances which, compared with distributions of randomly generated instances, are more difficult for linear programming algorithms to solve.

This work is in part a first step towards generating diverse MIP instances which vary features relevant to recent work on algorithm selection and runtime prediction. The long term goal of this research is to use synthetic data to augment real world test sets for experimental work in this domain, with the objective of enhancing insights into algorithm strengths and weaknesses.

It is a clear requirement when generating MIP instances that the root node LP relaxation is feasible and bounded. Without this property, test instances would be solved at the root node, without the need to branch or generate cutting planes, thereby failing to test any relevant parts of the solver. The correctness and completeness guarantees of the generator developed in this paper would allow for controlled variation of MIP features while avoiding such trivial and uninformative test cases.

The method used to develop the controllable generator for LPs can potentially be generalised to other optimisation problems. By defining an appropriate constructor, an alternative encoding for a problem space can be introduced to restrict generation and search to a specific problem class. Design of a generator for instances in this encoding should then aim to produce feature variation appropriate to the instance class. In future work we will be applying these ideas to generate new test instances for specific combinatorial optimisation problems, as well as MIP.

Prior to extending the work in these directions however, there are a number of limitations that will need to be addressed. Although local search performed well for the search objectives considered here, it is likely that generating instances with more complex characteristics will require global search algorithms. Developing operators for such algorithms, and proving they connect the search space, will be significantly easier using the instance constructor developed here. The alternative encoding introduced for instance data removes the requirement for repair heuristics to maintain a given instance property and makes it easy to verify that the search space is connected.

The effects of representation redundancy in the encoding space may need to be considered when developing these search algorithms. Further analysis will be required to understand the conditions where this occurs and what impact the imbalance has on the search procedure. An algorithm to produce alternative encodings may be required to transition between alternative representations in order to maintain search consistency. The resulting convergence of these search strategies, dependent on both the operator set and objective function, is an open problem for further theoretical analysis.

Finally, while the controllable generator developed in this paper is able to produce diverse instances with regard to the features considered, it is not yet sufficient to produce very challenging instances without the use of local search. This implies that additional features need to be considered in the generator design in order to produce the range of data required to find instances which are difficult to solve. The challenge here is to devise suitable new features that adequately capture what makes the target problem challenging for different solvers, as we have done for various other problems from optimization and other fields [27]. We refer the interested reader to our online tool MATILDA (available at https://matilda.unimelb.edu.au) for more details about the opportunities for understanding strengths and weaknesses of solvers via instance space analysis if diverse and challenging test instances can be generated.

Acknowledgements The authors would like to thank the anonymous referees from Mathematical Programming Computation whose thorough comments significantly improved the focus and quality of this work.

References

1. Asahiro, Y., Iwama, K., Miyano, E.: Random generation of test instances with controlled attributes. DIMACS Ser. Discrete Math. Theor. Comput. Sci. **26**, 377–393 (1996)
2. Bischl, B., Kerschke, P., Kotthoff, L., Lindauer, M., Malitsky, Y., Fréchette, A., Hoos, H., Hutter, F., Leyton-Brown, K., Tierney, K., Vanschoren, J.: ASlib: a benchmark library for algorithm selection. Artif. Intell. **237**, 41–58 (2016)
3. Bixby, R.E.: A brief history of linear and mixed-integer programming computation. Documenta Mathematica—Extra Volume ISMP pp. 107–121 (2012)
4. Bowly, S.: simonbowly/lp-generators: v0.2-beta (Version v0.2-beta). Zenodo. http://dx.doi.org/10.5281/zenodo.1220448 (2018)
5. Chakraborty, S., Choudhury, P.P.: A statistical analysis of an algorithm's complexity. Appl. Math. Lett. **13**(5), 121–126 (2000)
6. Cotta, C., Moscato, P.: A mixed evolutionary-statistical analysis of an algorithm's complexity. Appl. Math. Lett. **16**(1), 41–47 (2003)
7. Culberson, J.: Graph Coloring Page. http://webdocs.cs.ualberta.ca/~joe/Coloring/ (2010). Accessed 03 April 2017
8. Drugan, M.M.: Instance generator for the quadratic assignment problem with additively decomposable cost function. In: 2013 IEEE Congress on Evolutionary Computation, pp. 2086–2093. IEEE (2013)
9. Gao, W., Nallaperuma, S., Neumann, F.: Feature-based diversity optimization for problem instance classification. In: International Conference on Parallel Problem Solving from Nature, pp. 869–879. Springer (2016)
10. Gleixner, A., Hendel, G., Gamrath, G., Achterberg, T., Bastubbe, M., Berthold, T., Christophel, P.M., Jarck, K., Koch, T., Linderoth, J., Lübbecke, M., Mittelmann, H.D., Ozyurt, D., Ralphs, T.K., Salvagnin, D., Shinano, Y.: MIPLIB 2017. https://miplib.zib.de/ (2018). Accessed 30 July 2019
11. Hall, N.G., Posner, M.E.: The generation of experimental data for computational testing in optimization. In: Experimental Methods for the Analysis of Opimization Algorithms, pp. 73–101. Springer, Berlin Heidelberg (2010)
12. Hill, R., Moore, J., Hiremath, C., Cho, Y.: Test problem generation of binary knapsack problem variants and the implications of their use. Int. J. Oper. Quant. Manag. **18**(2), 105–128 (2011)
13. Hill, R.R., Reilly, C.H.: The effects of coefficient correlation structure in two-dimensional knapsack problems on solution procedure performance. Manage. Sci. **46**(2), 302–317 (2000)
14. Hooker, J.N.: Needed: an empirical science of algorithms. Oper. Res. **42**(2), 201–212 (1994)
15. Hooker, J.N.: Testing heuristics: we have it all wrong. J. Heuristics **1**(1), 33–42 (1995)
16. Hutter, F., Hoos, H.H., Leyton-Brown, K., Stützle, T.: ParamILS: an automatic algorithm configuration framework. J. Artif. Intell. Res. **36**(1), 267–306 (2009)
17. Hutter, F., Xu, L., Hoos, H.H., Leyton-Brown, K.: Algorithm runtime prediction: methods & evaluation. Artif. Intell. **206**(1), 79–111 (2014)
18. Kadioglu, S., Malitsky, Y., Sellmann, M., Tierney, K.: ISAC-instance-specific algorithm configuration. ECAI **215**, 751–756 (2010)
19. Klingman, D., Napier, A., Stutz, J.: NETGEN: a program for generating large scale capacitated assignment, transportation, and minimum cost flow network problems. Manage. Sci. **20**(5), 814–821 (1974)
20. Koch, T., Achterberg, T., Andersen, E., Bastert, O., Berthold, T., Bixby, R.E., Danna, E., Gamrath, G., Gleixner, A.M., Heinz, S., Lodi, A., Mittelmann, H., Ralphs, T., Salvagnin, D., Steffy, D.E., Wolter, K.: MIPLIB 2010: mixed integer programming library version 5. Math. Program. Comput. **3**(2), 103–163 (2011)
21. Leyton-Brown, K., Nudelman, E., Shoham, Y.: Empirical hardness models: methodology and a case study on combinatorial auctions. J. ACM **56**(4), 1–52 (2009)
22. Lin, S., Kernighan, B.W.: An effective heuristic algorithm for the traveling-salesman problem. Oper. Res. **21**(2), 498–516 (1973)

23. McGeoch, C.C.: Feature article—toward an experimental method for algorithm simulation. INFORMS J. Comput. **8**(1), 1–15 (1996)
24. Pilcher, M.G., Rardin, R.L.: Partial polyhedral description and generation of discrete optimization problems with known optima. Nav. Res. Logist. (NRL) **39**(6), 839–858 (1992)
25. Rice, J.R.: The algorithm selection problem. Adv. Comput. **15**, 65–118 (1976)
26. Smith-Miles, K., Bowly, S.: Generating new test instances by evolving in instance space. Comput. Oper. Res. **63**, 102–113 (2015)
27. Smith-Miles, K., Lopes, L.: Measuring instance difficulty for combinatorial optimization problems. Comput. Oper. Res. **39**, 875–889 (2012)
28. Todd, M.J.: Probabilistic models for linear programming. Math. Oper. Res. **16**(4), 671–693 (1991)
29. Van Hemert, J.I.: Evolving combinatorial problem instances that are difficult to solve. Evol. Comput. **14**(4), 433–462 (2006)
30. Xu, L., Hutter, F., Hoos, H.H., Leyton-Brown, K.: Hydra-MIP: automated algorithm configuration and selection for mixed integer programming. In: RCRA Workshop on Experimental Evaluation of Algorithms for Solving Problems with Combinatorial Explosion at the International Joint Conference on Artificial Intelligence (IJCAI), pp. 16–30 (2011)