Mathematical Programming Computation manuscript No.
(will be inserted by the editor)

Generation techniques for linear and integer programming instances with controllable properties

Simon Bowly * · **Kate Smith-Miles** ·
Davaatseren Baatar · **Hans Mittelmann**

Received: date / Accepted: date

Abstract This paper addresses the problem of generating synthetic test cases for experimentation in linear and mixed integer programming. We propose a generation framework to shift instance generation and search processes to an alternative encoded space. The framework is applied to produce 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 random generation and search algorithms in this framework generate only instances with this property.

Our results demonstrate that controlled random generation and instance space search using this method achieves feature diversity more effectively than using a direct representation. Furthermore, local search algorithms in the encoded space are able to increase the difficulty of generated instances for linear programming algorithms. This research opens further questions as to the suitability of various search algorithms for targeted instance design, convergence of search algorithms in instance space, and appropriate predictive features for LP and MIP algorithm performance.

* Corresponding author.

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

S. Bowly · K. Smith-Miles · D. Baatar
School of Mathematical Sciences
Monash University, Clayton VIC 3800, Australia
E-mail: simon.bowly@monash.edu

K. Smith-Miles
E-mail: kate.smith-miles@monash.edu

D. Baatar
E-mail: davaatseren.baatar@monash.edu

H. Mittelmann
School of Mathematical and Statistical Sciences
Arizona State University, Tempe, AZ 85287-1804, U.S.A.
E-mail: mittelmann@asu.edu

Keywords Linear programming · Mixed integer programming · Instance generation

Mathematics Subject Classification (2000) 90C05 · 90C10 · 90C11 · 68W40 · 90-08

1 Introduction

The success of empirical methods for algorithm 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 [1, 2]. MIPLIB [3] 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, without necessarily aiming to ensure that these instances are as diverse as possible in a way that support insights into algorithm strengths and weaknesses.

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 [4], and are not very diverse in measured features [5, 6]. Experimental hypothesis testing requires data where test parameters are appropriately varied and other influences are randomised [1, 7, 8]. Consequently, Hooker [9] advocated for the use of highly parameterised generators to produce appropriately controlled data for experimentation.

For various combinatorial optimisation problems - quadratic assignment [10], multidimensional knapsack [11], graph colouring [12] and boolean satisfiability [4] - 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 [6]. It is somewhat surprising then that the same attention has not yet been paid to producing new test instances for mixed integer programming, especially given the recent efforts to develop portfolio based and automatically configured MIP solvers [13, 14], and the need for representative test instances in the tuning of such strategies [15, 16]. Success in this area reinforces the usefulness of generating test data with controlled variation of all features of interest.

Existing generators for linear and integer programming instances may not be general enough for experimentation in MIP. The NETGEN generator [17], and its successor MNETGEN, produce parameterised linear programming (LP) instances, but is targeted specifically at multicommodity flow, transport and assignment problems. The parameters used are thus appropriate to the underlying network, not the LP feasible set or solution. Pilcher and Rardin [18] define a generator for pure integer programming problems with a known partial polytope by

introducing random cuts. However, the implementation is restricted to traveling 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 control a generator to produce instances with varying properties of interest. 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 control explicitly, but can be attained through an iterative search process that successively modifies an instance until it achieves the desired properties [5]. While such instance space search techniques are generally more computationally intensive than random generators, they do provide a reliable method for producing instances with specific target characteristics.

The most common search methods in use for this application are evolutionary algorithms. Chakraborty and Choudhury [19] and Cotta and Moscato [20] 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 [21] and diversity of measured features [5, 22]. 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 thus focuses on developing new instance generation techniques for linear and mixed integer programming test instances with controllable properties. We present a framework which allows only feasible, bounded LP instances to be generated, while controlling properties of the constraint structure and optimal solution. The addition of integrality constraints makes the results appropriate for use in feature-based models for MIP algorithm performance. Guaranteeing feasibility and boundedness of the linear programming relaxation ensures relatively trivial instances which do not challenge MIP solver components are not generated. The framework also incorporates methods for using iterative search approaches to find instances which are difficult to design or rarely produced by the generator.

The remainder of the paper is organised as follows: in Section 2 we describe the LP and MIP features that we wish to control, and the framework for how we control the generation of instances is presented in Section 3. Since the framework involves generating instances in a parameterised space before mapping to the LP space, we describe these two main steps in Sections 4 and 5.1. We then describe how local search can be used to improve the diversity of the generated instances in Section 5.2. Results for random generation and search are presented in Section 6.1 for feature space, and Section 6.2 for LP and MIP algorithm performance space, before conclusions are drawn in Section 7.

2 Linear Programming Metadata

We focus initially on LP instance generation in this work for several reasons. Firstly, solving LP relaxations can become a bottleneck for solving MIP problems as the size of models increases [23]. Indeed, there are examples in the MIPLIB2010 test set [3] for which the LP relaxation takes a long time to solve. Furthermore, each branching step requires re-optimisation of the LP with additional constraints,

which can take a significant amount of time in some cases. Designing LP instances for which the re-optimisation task is difficult is likely to produce MIP cases which are computationally expensive to solve. Secondly, features of the LP relaxation optimal solution are commonly used in building models for algorithm selection. For example, Hutter et al. [14] use measures of relaxation solution fractionality to predict algorithm runtime, Khalil et al. [24] use integrality slack distances to select branching variables, and Liberto et al. [25] use measures of active constraints to select heuristics. Thirdly, restricting generation and search to only produce feasible, bounded LP instances refines our search space, preventing trivial cases from being produced which are not likely to be informative of MIP solver performance.

The LP generator produces instances in canonical form:

$$\begin{aligned} \max\ & c^T x \\ \text{s.t. } & Ax \leq b \\ & x \geq 0 \end{aligned} \tag{1}$$

where $A \in \boldsymbol{R}^{m \times n}$, $b \in \boldsymbol{R}^m$ and $c \in \boldsymbol{R}^n$.

We will also consider the primal problem and its dual in standard equality form for the purposes of specifying instance data:

$$\begin{aligned} \max\ & c^T x \\ \text{s.t. } & Ax + s = b \\ & x \geq 0,\ s \geq 0 \end{aligned} \tag{2}$$

$$\begin{aligned} \min\ & b^T y \\ \text{s.t. } & A^T y - r = c \\ & y \geq 0,\ r \geq 0 \end{aligned} \tag{3}$$

Throughout the paper we will consider an optimal solution of a linear programming problem instance to be a complete specification $(\hat{x}, \hat{s}, \hat{y}, \hat{r})$ of an optimal solution.

Table 1 provides features we consider for measuring properties of linear programming instances. Features used in this work have been previously used for performance prediction in MIP [14], combinatorial auctions [26] and set covering [27]. Note that the variable constraint graph (VC) is a bipartite graph where an edge exists between variable node i and constraint node j only if $a_{ij} \neq 0$.

To measure the computational effort required to solve the generated LPs, and their potential effect on difficulty of any resulting MIPs, we measure solver performance at the root node and for node re-optimisation after branching. The COIN LP solver is used to test primal simplex, dual simplex and barrier algorithms solving the generated root LP. For re-optimisation, each generated LP instance is converted to a pure integer program by adding integrality constraints to all variables. SCIP [28] with the SoPlex primal-dual LP solver is run to solve the root node and carry out full strong branching. The statistics collected from full strong branching measures the number of simplex iterations required to re-optimise all possible branch relaxations from the root node. Table 2 provides the algorithms and metrics considered.

Relaxation Features
Number of binding constraints at the LP optimum
Number of fractional primal variables at the LP optimum
Integer slack vector (minimum Manhattan distance of optimal point to an integral point)
Variable Constraint Graph Features
Degree sequence of variable nodes in VC (min/mean/max/stddev)
Degree sequence of constraint nodes in VC (min/mean/max/stddev)
Coefficient Value Features
Coefficient statistics $\{a_{ij} \mid a_{ij} \neq 0\}$ (min/mean/max/stdev)
Right hand side statistics $\{b_j\}$
Objective coefficient statistics $\{c_i\}$
Row-normalised coefficient statistics $\{\frac{a_{ij}}{b_j} \mid b_j \neq 0\}$ (min/mean/max/stdev)
Column-normalised coefficient statistics $\{\frac{a_{ij}}{c_i} \mid c_i \neq 0\}$ (min/mean/max/stdev)
Constraint degree-normalised rhs statistics $\{\frac{b_j}{u_j}\}$ (min/mean/max/stdev)
Variable degree-normalised objective statistics $\{\frac{c_i}{v_i}\}$ (min/mean/max/stdev)

Table 1: Linear programming features.

Algorithm	Metric
Primal simplex	Number of iterations to solve LP
Dual simplex	Number of iterations to solve LP
Barrier	Number of floating point operations to solve LP
Simplex re-optimisation	Number of iterations to re-optimise per branch variable of IP

Table 2: LP algorithm performance metrics.

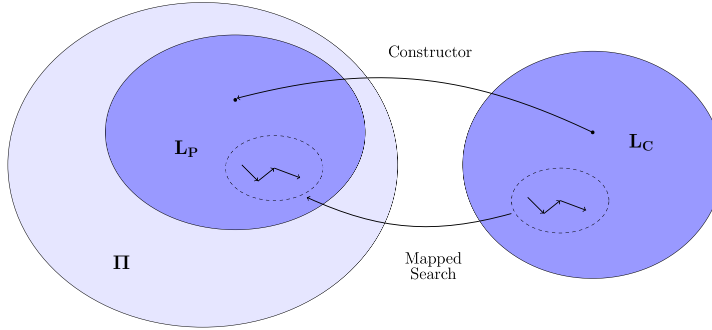


Fig. 1: Constructor mapping shifting an instance generation and search problem to an encoded representation.

3 Generation Framework

In this section we present a general framework for controlled random generation and targeted search in a restricted instance space. In particular we aim to combine the theoretical ideals of efficiently generating instances with known solutions [29] and explicitly controlling decision properties such as feasibility [4], with experimental ideals of varying structural features relevant to a given experiment [8, 11]. The framework allows the generation method to be extended with search techniques to find instances with target characteristics [5, 21].

Application of the framework to a new problem class requires defining an alternative encoding for instances of that class and a construction algorithm to produce instances from the encoded form. A parameterised random generation strategy generates encoded instances while varying appropriate features, while neighbourhood operators make modifications in the encoding space as part of search processes. The instance generation and search problem is thereby shifted into the encoded space, and the constructor algorithm maps the results onto the target problem class (see figure 1). In general we aim to guarantee that generation algorithms are *correct*, in that they produce only instances from the target class, and *complete* in that any instance in the target class can be produced.

In this case the constructor maps an encoded form for LPs to the class of feasible, bounded LP instances such that their optimal solution is known. The construction process is efficient since LP requires only a linear sized optimality proof. Introducing this mapping produces much simpler conditions to satisfy feasibility and boundedness compared to a direct representation of the LP data. The resulting random generation algorithms and search operators do not produce instances outside the target class, potentially improving the efficiency of instance generation.

Problem space Let $\boldsymbol{\Pi}$ be an optimisation problem, $\boldsymbol{P}$ be a subclass of $\boldsymbol{\Pi}$ over which algorithm performance is to be studied, $\boldsymbol{A}$ be a portfolio of algorithms, $\boldsymbol{F}$ be a feature set which applies to $\boldsymbol{P}$. A test case generator for $\boldsymbol{P}$ is required to output, nondeterministically, instances of $\boldsymbol{P}$ according to some distribution. The generator should be capable of producing any instance of $\boldsymbol{P}$, although not necessarily with equal probability.

In this paper, we develop a test case generator for feasible, bounded linear programs, as a subclass of linear programs in general.

Instance language encoding Let Σ be a finite alphabet of symbols, and the language Σ^* be the set of all finite length strings of the symbols in Σ. All instances of $\boldsymbol{\Pi}$ have an encoding as a string $e \in \Sigma^*$. Now define an acceptor Γ which, given a string $e \in \Sigma^*$, returns whether e represents a valid instance of $\boldsymbol{P}$. $\boldsymbol{P}$ is then formally defined as a language $L_P \subset \Sigma^*$ for which Γ responds 'yes' to the decision problem "is e a member of $\boldsymbol{P}$?" for all e in L_P (alternatively $\Gamma(e) = 1 \,\forall\, e \in L_P$). The generator for $\boldsymbol{P}$ is required to output strings in L_P according to some distribution.

Linear programming instances are encoded over the real numbers (Σ) as a tuple $e = (A, b, c) \in \Sigma^*$. The rational numbers can be considered a finite alphabet for the purposes of computation, as they are encoded as bit strings of finite length. The acceptor Γ must then be an algorithm for which $\Gamma(e) = 1$ when e represents a feasible, bounded linear program (has an optimal solution) and 0 otherwise. In this way the acceptor defines L_P to be the set of all feasible, bounded linear programming instances.

Intermediate language When the decision-making process of Γ is complex (requires some algorithm), it will become useful to split the generator into two components. We first define an intermediate language L_C. The constructor C accepts strings in L_C as input, and outputs only strings in L_P. The construction process is deterministic. This approach has the effect of restricting the instances produced

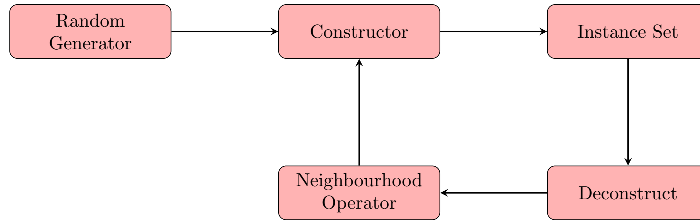


Fig. 2: Constructor implementation for random generation and search operators.

to precisely those required to represent the subclass $\boldsymbol{P}$. The generator G nondeterministically outputs strings in L_C to be used by the constructor.

The intermediate language L_C applied in this generator encodes a linear programming instance as a tuple (A, α, β), where A matches its definition in L_P and α, β together encode a complete optimal primal-dual solution to the instance. The generator produces constraint matrices paired with an optimal solution, while the constructor recovers a linear program with the given constraint matrix for which the generated solution is optimal.

As well as solution information, strings in L_C can include parameters to aid in designing instances with required feature values. This approach guides the design of the generator in terms of the feature set, supporting experimental design protocols. A generator should have the ability to generate instances conforming to a target distribution of output features as required by a given experiment.

Instance space search The intermediate language also supports design of search algorithms in the problem space. As shown in figure 2, any neighbourhood operator to be applied in searching the problem space should produce strings in L_C. This guarantees that the operator, like the generator, produces strings in L_P after construction. Appropriate choice of L_C makes it easier to design neighbourhood operators which do not violate the conditions of the subclass L_P, obviating the need to apply repair heuristics which may introduce bias.

To search the space of feasible linear programs, a neighbourhood (or metaheuristic) search operator would modify (A, α, β) in L_C. The constructor then guarantees that the new instances returned are feasible and bounded. Applying search operators instead to the direct encoding (A, b, c) has the potential to create infeasible or unbounded instances.

Search criteria For a given experiment or exploratory study, one may define measures for the sufficiency of the instance set in terms of algorithm performance and feature distributions. A search algorithm, using neighbourhood operators as explained above, optimises the sufficiency metric by generating new instances. Figure 3 gives an iterative framework for updating the instance set using search results in order to meet the sufficiency requirement.

It is possible that a random generation approach may need search extensions to completely represent a feature space. For an instance set which does not cover the full feature range the objective should favour instances which increase feature diversity. Previous work has used minimisation of distance to target points in

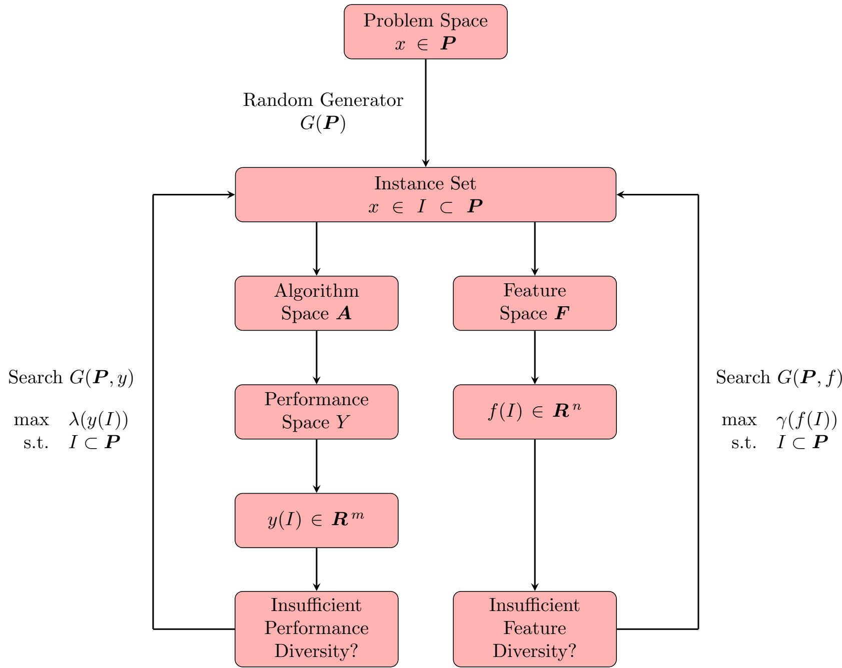


Fig. 3: Optimisation framework for random generation and search to improve test set diversity.

feature space [5] and measures of instance contribution to feature diversity [22] as objectives.

For an instance set which does not contain enough hard instances, or enough instances which are discriminating of algorithm performance, the objective function may be based on performance metrics. This approach has been applied previously for TSP [21]. We note that, under the assumption that instance features explain algorithm performance, a lack of performance diversity indicates that the current feature set is insufficient. The main application of this type of search then should be to produce instances which elicit different algorithm performance characteristics in order to develop new features. As with real world data sets, choosing experimental data based on performance of current algorithms is likely to lead to bias, so we note that such an approach should be specific to exploratory analysis.

4 Linear Programming Constructor

As in section 3, $\boldsymbol{\Pi}$ is the class of all linear programs, and $\boldsymbol{P}$ is the set of all feasible, bounded linear programs. Strings (A, b, c) in the language Σ^* define linear programs in canonical form. The language L_P is then the subset of strings for which an optimal solution can be found for the corresponding instance.

Feasibility cannot be determined directly from the string (A, b, c). To define the class, we require an acceptor which solves the given linear program, responding 'yes' if it terminates at an optimal point (as opposed to proof of infeasibility or unboundedness). Therefore a simple generation routine with $L_C = L_P$ is unlikely to suffice for this application. Instead, we define a new encoding of the linear program, which stores the constraint coefficients along with a complete optimal solution. The constructor processes this alternative encoding to produce a feasible, bounded linear program.

The encoding stores the constraint matrix A of the primal canonical form problem given in equation (1). An optimal basic feasible solution is encoded using a basis B for the primal problem and its complement N for the dual. The constructor input language enforces that only those primal problem variables in B, and dual problem variables in N are non-zero, ensuring the complementarity relation is always satisfied. Ensuring feasibility of the given solution is then the only requirement to design an instance for which it is optimal.

The intermediate language L_C for the constructor is the set of all tuples (n, m, A, α, β) which satisfy

$$\begin{array}{ll} n, m \in \boldsymbol{N} & \\ A \in \boldsymbol{R}^{m \times n} & \\ \alpha \in \boldsymbol{R}^{m+n} & \alpha_i \geq 0 \\ \beta \in \boldsymbol{B}^{m+n} & \sum \beta_i = m. \end{array} \tag{4}$$

Membership of L_C is easy to determine using equation (4). Given an input in L_C, the constructor will return a feasible, bounded linear program with n variables and m constraints in canonical form (A, b, c).

Algorithm 1 Constructor for feasible, bounded linear programs.

Require: $(n, m, A, \alpha, \beta) \in L_C$
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) linear program in canonical form

Proposition 1 *Outputs of the constructor are feasible, bounded linear programs with optimal basic feasible solution $(\hat{x}, \hat{s})$ for the primal and $(\hat{y}, \hat{r})$ for the dual problem.*

Proof Consider the equality form of the primal problem P given in equation (2), and the corresponding equality form of the dual problem D given in equation (3). Non-negativity constraints for $(\hat{x}, \hat{s})$ and $(\hat{y}, \hat{r})$ are satisfied since α is non-negative. $(\hat{x}, \hat{s})$ is feasible to P, by definition of b and $(\hat{y}, \hat{r})$ is feasible to D, by definition of

c in algorithm 1. Both P and D are therefore feasible. It follows from weak duality that both are bounded. Since the primal problem is feasible and bounded, it has at least one basic feasible solution which is optimal.

$(\hat{x}, \hat{s})$ is a basic feasible solution to P since $\sum \beta = m$ and $(\hat{x}, \hat{s})$ are nonzero only for $\beta_i = 1$. Similarly for D. Furthermore, $(\hat{x}, \hat{s})$ and $(\hat{y}, \hat{r})$ satisfy the complementary slackness conditions, since

$$x_i r_i = \beta_i (1 - \beta_i) \alpha_i^2 = 0 \quad \forall\, i = 1 \ldots n$$

$$s_j y_j = \beta_{j+n} (1 - \beta_{j+n}) \alpha_{j+n}^2 = 0 \quad \forall\, j = 1 \ldots m$$

It follows from strong duality that $(\hat{x}, \hat{s})$ and $(\hat{y}, \hat{r})$ are optimal basic feasible solutions to P and D respectively.

Proposition 2 *Any feasible, bounded linear program has at least one encoding in L_C.*

Proof Any feasible, bounded LP instance in standard form (equation (2)) P has at least one optimal basic feasible solution, defined by the basis B containing m elements. Solving such an instance using the simplex algorithm recovers an optimal basis B, along with the primal and dual solution information (x, y, r, s). Define β such that $\beta_i = 0$ for all $i \in B$ and 0 otherwise. Generate α using the following algorithm:

for $i = 1 \ldots n$ **do**
  if $i \in B$ **then** $\alpha_i := \hat{x}_i$ **else** $\alpha_i := \hat{r}_i$ **end if**
end for
for $j = 1 \ldots m$ **do**
  if $j + n \in B$ **then** $\alpha_{j+n} := \hat{s}_j$ **else** $\alpha_{j+n} := \hat{y}_j$ **end if**
end for

Given that a basis of solution contains m elements, and basic feasible solutions must be non-negative, we have $\alpha \geq 0$ and $\sum \beta = m$. The matrix A is not changed in the encoded form. Therefore $e = (n, m, A, \alpha, \beta)$ defines an encoded linear program in L_C, for which $P = C(e)$. Therefore any feasible, bounded linear program P has a corresponding encoding in L_C and is produced by the constructor given appropriate inputs satisfying equation (4).

Proposition 1 implies that algorithm 1 is correct in that all linear programs produced by the constructor are feasible and bounded. Proposition 2 implies that algorithm 1 is complete in that any feasible, bounded linear program has an equivalent set of constructor inputs which will produce it. It follows that L_C is a complete representation of L_P, with feasibility and boundedness enforced by simple conditions on the string (n, m, A, α, β). Therefore a generating machine which can output any string in L_C creates a generator for all of $\boldsymbol{P}$.

The constructor is guaranteed to produce a unique output instance for a given encoded input. However, the map is not one-to-one with respect to the set of feasible, bounded LPs encoded in the form (A, b, c). Any instance with multiple optimal bases for the primal or dual problem will have multiple encodings. This can occur where there exist feasible entering and leaving variables from the optimal basis, or when the solution basis specified in the encoding corresponds to a singular submatrix of constraints. As a result a generation algorithm which produces uniformly random encoded strings may favour instances with multiple representations.

5 Generation of Encoded Instances

5.1 Random Generator

The generator algorithm produces strings $(n, m, A, \alpha, \beta) \in L_C$. Appropriate parameters are given to control desired features within the randomisation scheme. After specifying the problem size, the remaining parameters are size-independent, intended to capture instance structure.

The generation process is carried out in two stages. The first is concerned with generating the optimal solution for the constructed instance, including primal and dual solution values, primal constraint slacks and dual constraint surpluses. The generator outputs α, β vectors which define a valid, 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 2 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 strictly 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.

The second stage generates the constraint matrix A. 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 3 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. Any unconnected nodes remaining after this process are connected to prevent the algorithm producing empty constraints or unbounded variables. Nonzero values in the constraint matrix are then drawn from a log-normal distribution.

Given the choice of primal and slack variable indices in algorithm 2, 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. Furthermore, given the binary choice of values in β, $\beta \in \boldsymbol{B}^{n+m}$ and $\sum_{i=1}^{n+m} \beta_i = |P_{opt}| + |S_{opt}| = k + (m - k) = m$ as required by the constructor input language L_C.

Elements of $\{\alpha_i \mid i \leq n\}$ are first drawn from $\lceil X_1 \rceil \in [1, \infty)$, since the log-normal distribution is strictly positive and 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 $\alpha_i > 0$.

Algorithm 2 Solution generator.

Require: $n \in [1, \infty]$, $m \in [1, \infty]$, $\gamma \in [0, 1]$, $\lambda \in [0, 1]$, $a \in (0, \infty)$, $\mu_p \in (-\infty, \infty)$, $\sigma_p \in (0, \infty)$, $\mu_s \in (-\infty, \infty)$, $\sigma_s \in (0, \infty)$

1: $X_1 \sim \text{LogNormal}(\mu_p, \sigma_p)$,
2: $X_2 \sim \text{Beta}(a, a)$,
3: $X_3 \sim \text{LogNormal}(\mu_s, \sigma_s)$.
4: $k \leftarrow [\gamma \min(n, m)]$
5: $P_{opt} \leftarrow$ Choose k indices from $\{1 \ldots n\}$ without replacement.
6: $S_{opt} \leftarrow$ Choose $m - k$ indices from $\{1 \ldots m\}$ without replacement.
7: $P_{frac} \leftarrow$ Choose $[\lambda n]$ indices from $\{1 \ldots n\}$ without replacement.
8: **for** $i = 1 \ldots n$ **do**
9: **if** $i \in P_{opt}$ **then** $\beta_i := 1$ **else** $\beta_i := 0$ **end if**
10: **if** $i \in P_{frac}$ **then** $\alpha_i := \lceil X_1 \rceil - X_2$ **else** $\alpha_i := \lceil X_1 \rceil$ **end if**
11: **end for**
12: **for** $j = 1 \ldots m$ **do**
13: **if** $i \in S_{opt}$ **then** $\beta_{n+j} := 1$ **else** $\beta_{n+j} := 0$ **end if**
14: $\alpha_{n+j} := X_3$
15: **end for**
16: **return** (α, β)

Algorithm 3 Constraint generator.

Require: $n \in [1, \infty]$, $m \in [1, \infty]$, $\rho \in (0, 1]$, $\hat{x}$, $\hat{y}$, $p_v \in [0, 1]$, $p_c \in [0, 1]$, $\mu_A \in (-\infty, \infty)$ $\sigma_A \in (0, \infty)$

1: **while** $|E(VC)| < \rho mn$ **do**
2: Increment the degree of variable node i with maximum $\frac{p_v}{\sum_i v_i} v_i + U(0, 1)$
3: Increment the degree of constraint node j with maximum $\frac{p_c}{\sum_j u_j} u_j + U(0, 1)$
4: **end while**
5: **for** $i = 1 \ldots n$ **do**
6: **for** $j = 1 \ldots m$ **do**
7: Add edge (i, j) to VC with probability $\frac{v_i u_j}{|E(VC)|}$
8: **end for**
9: **end for**
10: **while** $\min(v_i, u_j) = 0$ **do**
11: Choose i from variable vertices with degree 0, or randomly if all $v_i > 0$
12: Choose j from constraint vertices with degree 0, or randomly if all $u_j > 0$
13: Add edge (i, j) to VC
14: **end while**
15: **for** $(i, j) \in E(VC)$ **do**
16: $a_{ji} := N(\mu_A, \sigma_A)$
17: **end for**
18: **return** A

Therefore $\alpha \geq 0$ as required by L_C. Solution pairs (α, β) produced by the generator satisfy the input language constraints. Including the constraint matrix generated by algorithm 3, gives a string $(A, \alpha, \beta) \in L_C$, so the generator algorithm is correct.

As above, $E[|P_{opt}|] = \gamma \min(n, m)$ and $E[|S_{opt}|] = m - \gamma \min(n, m)$. 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 $E[I] = \lambda E[|P_{opt}|] = \lambda\gamma \min(n, m)$.

Fractional components for each fractional primal variable are drawn from a symmetric beta distribution with parameter a. This symmetric beta distribution is uniform for $a = 1$, centrally skewed for $a > 1$, and edge-skewed for $a < 1$. 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 and definition of the partial beta function, the mean fractionality as a function of the parameter a for the symmetric beta distribution is calculated as:

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

Finally, we have

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

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 some capability to produce a given distribution of features by distributing 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. 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 a more complex variation, however approximately uniform variation can be achieved experimentally by selecting parameter values from a log-normal distribution.

5.2 Local Search

While a well defined generator is theoretically capable of producing any instance in the defined search space, some instance classes are less likely to appear than others. Specifically, some instance feature values are highly correlated, and performance criteria are not addressed at all. As a result the test set will likely not be balanced across all feature ranges, and some feature values are likely to be missing entirely, as they appear with a very low probability. Search methods may be advantageous in addressing this deficiency.

Search is carried out in the L_C encoded space. Strings manipulated by the local search operators are passed to the constructor to produce new instances. An instance space search operator takes a string in L_C and returns a randomly modified string in L_C. Search operators should therefore maintain the invariants given in the constructor input conditions in order to ensure generated neighbours are members of L_P.

Consider the following collection of neighbourhood operators:

1. Swap an element in β
2. Scale an element in α by $\lambda \in \boldsymbol{R}^+$
3. Remove a non-zero element a_{ij}
4. Add a non-zero element a_{ij}
5. Scale an element a_{ij}

Operator 1 only swaps the 0/1 elements of β, maintaining the invariant $\sum \beta = m$. Operator 2 scales solution values by a positive multiplier, so will not violate the condition that $\alpha \geq 0$. Operators 3-5 manipulate entries in the constraint matrix, which has no restriction. This set of operators is therefore correct, in that the conditions of L_C required to produce feasible linear programming instances are never violated.

Clearly, with a finite number of applications of operator 1, a binary string β can be transformed into a binary string β' with the same number of ones. Similarly, scaling values in α by operator 2 and manipulating A by operators 3-5 can transform any (A, α) to an alternative (A', α'). Therefore any $I' = (A', \alpha', \beta')$, can be reached from any $I = (A, \alpha, \beta)$ in a finite number of steps of the above operators. The set of operators is therefore complete with respect to fixed size instances. A search algorithm using these neighbourhood operators is therefore capable of traversing the entire space, given enough time, and is guaranteed not to produce infeasible or unbounded LP relaxations.

6 Results

This section compares results using the constructor generation approach with a simple random generation method producing LP data directly. Random generation and search algorithms are applied to achieve diversity in feature values and performance characteristics for the generated instances.

Generation and search is implemented in Python 3.5.2, with a C++ extension using the Coin LP callable library to facilitate feature computation. The pseudo-random number generator included with the numpy library (version 1.12.1) for python is used wherever random values or search decisions are made. Generation, feature search and performance search results are reproducible for a given (32 bit) seed value for the numpy random generator. Coin LP 1.16.10 is used to test primal simplex, dual simplex and barrier solvers on generated instances. SCIP 3.2.1 with SoPlex 2.2.1 as the LP solver is used for re-optimisation tests. Computations are run on an Ubuntu 16.04 virtual machine with 4 virtual cores and 8GB virtual memory. The code required to reproduce the experiments in this section is available from the Zenodo repository at doi:10.5281/zenodo.556009 [30].

6.1 Feature Space

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 initial 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 variables and constraints, number of binding constraints, number of

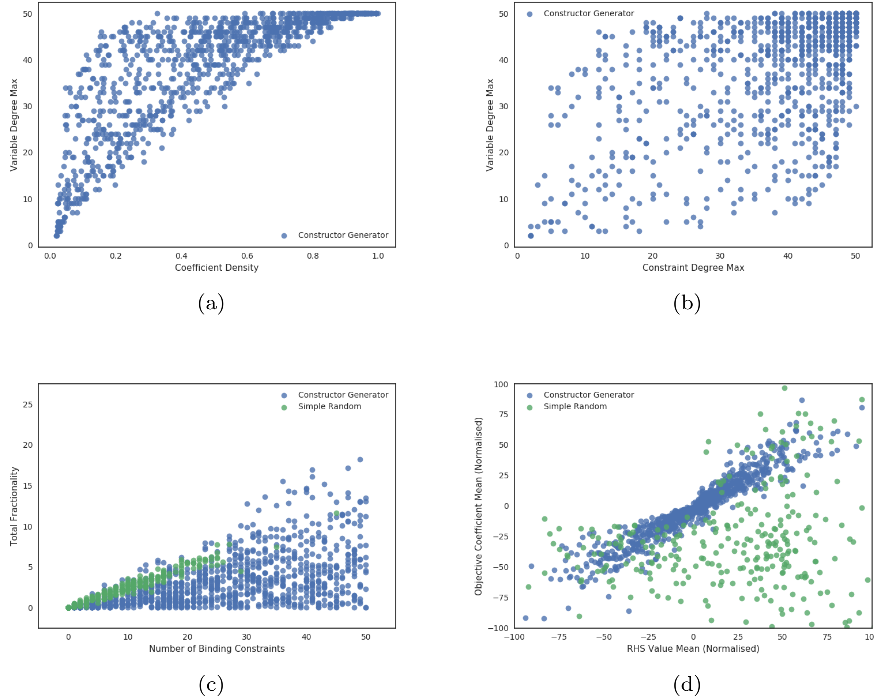


Fig. 4: Generated instances using solution construction and simple random generation. The simple random instances are shown only for spaces where their distributions differs from the constructor method.

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. Size-independent parameter distributions used to vary instance features are given in table 3.

We note that the results shown in figure 4 are just one example of a feature distribution which could be achieved. Altering the parameter input distributions changes the output feature distributions, producing varying results when required for a particular experiment. The approach for generating constraints can likely be further varied by introducing alternative graph structure generation approaches to that applied in algorithm 3.

We also build a simple random generator for comparison with the construction approach. This process uses the same algorithm to build constraint coefficient matrices, but randomly generates coefficients for the constraint bounds b and objective coefficients c instead of specifying an optimal solution. Mean and standard deviation values for b and c elements are chosen uniformly at random within the ranges we observe in the outputs of the constructor generator. Identical parameters are used in generation of the constraint matrix in each case. Only feasible and

Parameter	Value/Distribution
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 vs slack basis γ	$U(0, 1)$
Fractional primals λ	$U(0.1, 1.0)$
Beta fractionality a	$LN(-0.2, 1.8)$

Table 3: Generator parameters used in experiment.

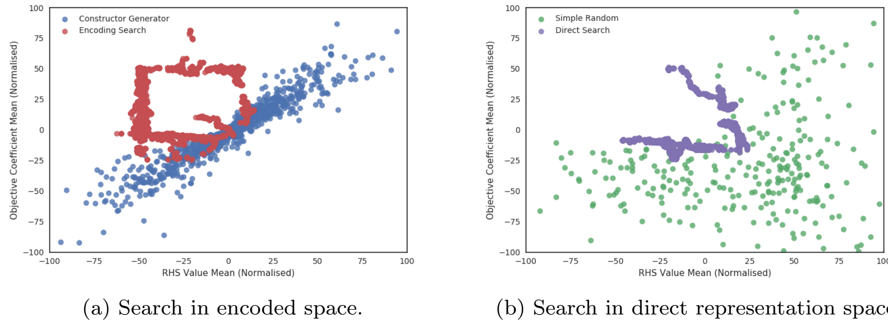
bounded instances generated using this method (40% of instances) are shown in the resulting plots.

Figures 4a and 4b show features of the constraint matrix which are explicitly varied. Results from the simple random approach are not shown, as the constraint matrix generation parameters are identical between the two methods. Degree distributions vary maximum values for variables and constraints within the bounds set by the bipartite graph edge density. Density is uniformly distributed in accordance with the input parameter. The construction of degree sequences is able to produce instances with high maximum variable degree and uniform constraint degree, and vice versa.

Figure 4c shows a relaxation solution feature space, indicating the trade-off between number of binding constraints and fractionality measures. The parameter controlling the ratio of primal and slack variables in the optimal basis is chosen uniformly, resulting in a uniform distribution of the number of binding constraints. Given that the size of the basis is fixed, the number of nonzero primal variables is also uniformly distributed. This sets an upper limit on the number of fractional variables which in turn bounds the total fractionality of the problem. As expected, instances generated using the simple random approach show very little variation in relaxation features. The constructor generator approach achieves significantly more diversity in the relaxation space, as it is able to directly parameterise the required features and vary them across their maximum relative ranges.

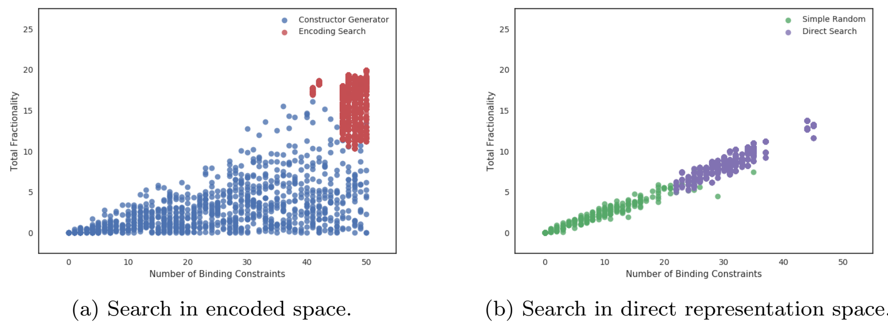
Figure 4d shows a coefficient statistic feature space, plotting the mean value of objective coefficients and constraint bounds. Both axes are normalised by the mean absolute value of constraint coefficients, to avoid easily producing instances with very large feature values simply by scaling the entire problem by a positive constant. While the constructor method captures most of the variation possible in these relaxation feature values, it introduces a correlation between values in the constraint upper bounds and objective coefficients. It is clear that the correlation is produced by the constructor given the calculation of b and c in algorithm 1. The simple random generator achieves more diversity in this space, however it also very rarely produces instances in the upper left quadrant of figure 4d, where constraint upper bounds are negative and objective coefficients are positive. Given the form of the primal problem (equation (1)), in this region the trivial solution is likely to be infeasible for both the primal and the dual. Search methods can be applied in the encoded and direct space to test whether it is possible to generate instances in this region using either representation.

Neighbourhood operators for local search are specified in section 5. For search in the direct representational space, the same operators are applied to the con-



(a) Search in encoded space. (b) Search in direct representation space.

Fig. 5: New instances generated by searching from random start locations towards the top left quadrant of objective/bound coefficient feature space.

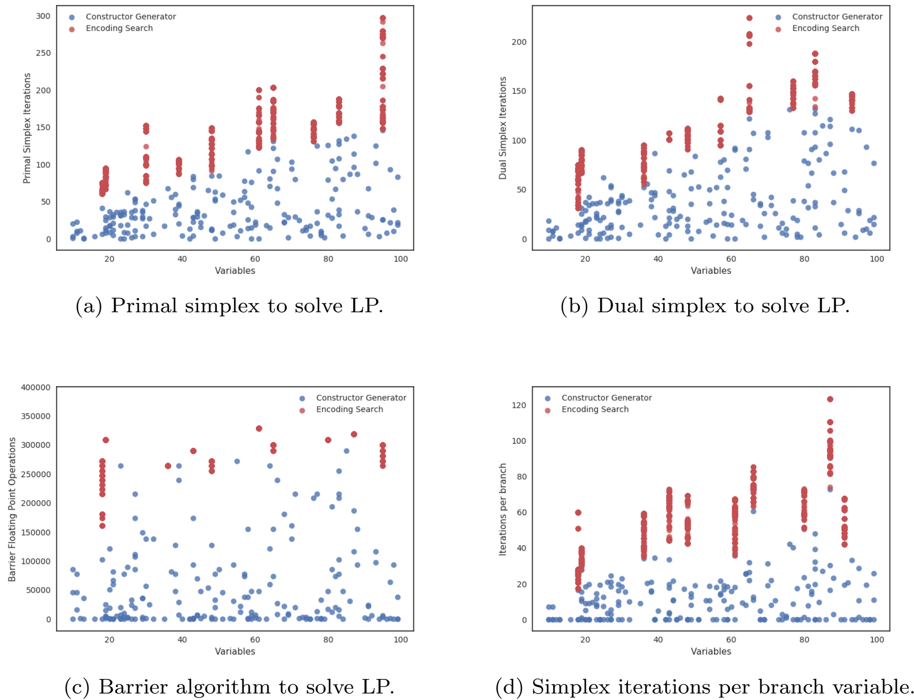


(a) Search in encoded space. (b) Search in direct representation space.

Fig. 6: New instances generated by searching from random start locations. The search aims to increase the number of binding constraints at the optimal point while improving diversity of the total fractionality feature.

straint coefficients. Further operators alter the objective coefficients and constraint bounds by scaling individual values. Any infeasible or unbounded instances are rejected as search candidates, as we aim to compare search effectiveness between the two approaches within the class of feasible, bounded problems. Search proceeds in a greedy local manner by generating a random neighbour at each step and accepting it only if it improves the target metric (distance to a target point in feature space). Each search starts at a random instance and is run for 1000 steps.

Figure 5 shows new instances generated by searching towards target points in the upper left quadrant of the space. It is clear that search in the constructor space is much more effective for this application. Successive local modifications to the solution encoding, which maintain feasibility and boundedness, can slowly converge over time to produce instances which are very rarely reached by the random generation method. Search in the direct representational space tends to frequently produce infeasible or unbounded instances, and stalls from many starting points.



(a) Primal simplex to solve LP.

(b) Dual simplex to solve LP.

(c) Barrier algorithm to solve LP.

(d) Simplex iterations per branch variable.

Fig. 7: New instances generated by searching in encoded space to increase an algorithm difficulty metric.

Allowing the search to generate infeasible instances, which are later recovered from, or applying repair heuristics to generated neighbours, could improve this behaviour. However, both approaches are likely to be much more computationally intensive or introduce biases.

Figure 6 shows new instances generated by searching for instances with more binding constraints and higher fractionality. We observe that search cannot overcome the deficiencies resulting from using a direct representation of LP data. Searching in the encoded space, however, can freely manipulate these features to reach a target point.

These results indicate that the constructor framework significantly improves our ability to control the distribution of relaxation features in the instance set. Moreover, deficiencies in diversity introduced by the encoded representation for other features can be effectively overcome using search. The combination of techniques is clearly more effective than attempting to apply search algorithms to overcome lack of diversity resulting from simple random generation.

6.2 Performance Space

While the generated instances are diverse in the feature classes we measure, performance metrics indicate that they may not be particularly difficult to solve. Figure 7 shows instances generated using the same size-independent generation parameters and uniformly distributed in size between 10 and 100 variables and constraints. Search is also applied in the encoded space to produce instances which are more challenging by a given performance metric. Each search begins at the hardest to solve generated instance in a given instance size range, and proceeds for 200 steps. At each step, the neighbouring instance is accepted if it takes more iterations for the target algorithm to complete. New instances are generated which are harder for primal and dual simplex algorithms to solve from scratch, and which are harder for branch reoptimisations.

Instance space search is able to produce harder instances than those produced by the generator for each algorithm tested. This indicates that the instances produced by the initial generation runs do not challenge the target algorithms sufficiently. Furthermore, if the feature set used to motivate the generator algorithm design contained good predictors of algorithm performance, we should expect a reasonable number of difficult instances to appear by generating instances with diverse features. Therefore it is likely that the feature set can be improved to better predict algorithm performance.

7 Conclusion

This paper proposes an instance generation framework which is applied to produce LP instances with controllable properties. In particular, the algorithm we develop varies features relevant to MIP experimentation. We have demonstrated that the generator is theoretically capable of producing any feasible bounded LP, given the right parameter choices. Local search methods can then be applied to generate instances with characteristics which are rarely produced by the generator, or are more difficult to solve. This work should be viewed as 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 is to use synthetic data to augment real world test sets for experimental work in this domain, with the objective of enhancing insights into MIP solver strengths and weaknesses.

Of course, the framework defined in this paper can be applied to generate instances for specific optimisation problems as well, restricting the types of constraints considered. 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 targets considered here, it is likely that achieving more complex instance characteristics will require global search or evolutionary algorithms. Designing operators for such algorithms will be significantly easier using the encoded LP

form. For evolutionary algorithms, for example, a row or column crossover scheme can maintain the constructor language invariant and hence reliably generate child instances of the target class. A similar approach applied to the original instance representation would likely require repair heuristics to maintain feasibility and boundedness of the generated children.

The effects of representation redundancy in the encoding space may need to be considered when developing further 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 random generator is able to produce diverse instances with regard to the features considered, it is not yet sufficient to produce very challenging instances. This implies that alternative 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 MIP, or any optimisation problem, challenging for different solvers or tricks employed by solvers.

References

1. John N. Hooker. Testing heuristics: We have it all wrong. *Journal of Heuristics*, 1(1):33–42, 1995.
2. Bernd Bischl, Pascal Kerschke, Lars Kotthoff, Marius Lindauer, Yuri Malitsky, Alexandre Fréchette, Holger Hoos, Frank Hutter, Kevin Leyton-Brown, Kevin Tierney, and Joaquin Vanschoren. ASlib: A benchmark library for algorithm selection. *Artificial Intelligence*, 237:41–58, August 2016.
3. Thorsten Koch, Tobias Achterberg, Erling Andersen, Oliver Bastert, Timo Berthold, Robert E. Bixby, Emilie Danna, Gerald Gamrath, Ambros M. Gleixner, Stefan Heinz, Andrea Lodi, Hans Mittelmann, Ted Ralphs, Domenico Salvagnin, Daniel E. Steffy, and Kati Wolter. MIPLIB 2010: Mixed integer programming library version 5. *Mathematical Programming Computation*, 3(2):103–163, 2011.
4. Yuichi Asahiro, Kazuo Iwama, and Eiji Miyano. Random generation of test instances with controlled attributes. *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, 26:377–393, 1996.
5. Kate Smith-Miles and Simon Bowly. Generating new test instances by evolving in instance space. *Computers and Operations Research*, 63:102–113, 2015.
6. Raymond R. Hill and Charles H. Reilly. The effects of coefficient correlation structure in two-dimensional knapsack problems on solution procedure performance. *Management Science*, 46(2):302–317, 2000.
7. Catherine C. McGeoch. Feature Article - Toward an Experimental Method for Algorithm Simulation. *INFORMS Journal on Computing*, 8(1):1–15, February 1996.
8. Nicholas G Hall and Marc E Posner. The Generation of Experimental Data for Computational Testing in Optimization. In *Experimental Methods for the*

Analysis of Opimization Algorithms, pages 73–101. Springer-Verlag, Berlin Heidelberg, 2010.
9. J. N. Hooker. Needed: An Empirical Science of Algorithms. *Operations Research*, 42(2):201–212, April 1994.
10. Mădălina M. Drugan. Instance generator for the quadratic assignment problem with additively decomposable cost function. In *2013 IEEE Congress on Evolutionary Computation*, pages 2086–2093. IEEE, 2013.
11. RR Hill, JT Moore, C Hiremath, and YK Cho. Test problem generation of binary knapsack problem variants and the implications of their use. *Int J Oper Quant Manag*, 18(2):105–128, 2011.
12. Joseph Culberson. Graph Coloring Page, August 2010. `https://webdocs.cs.ualberta.ca/~joe/Coloring` [Accessed 04/03/2017].
13. Lin Xu, Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. 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)*, pages 16–30, 2011.
14. Frank Hutter, Lin Xu, Holger H. Hoos, and Kevin Leyton-Brown. Algorithm runtime prediction: Methods & evaluation. *Artificial Intelligence*, 206(1):79–111, 2014.
15. John R Rice. The Algorithm Selection Problem. *Advances in Computers*, 15: 65–118, 1976.
16. Frank Hutter, Holger H. Hoos, Kevin Leyton-Brown, and Thomas Stützle. ParamILS: an automatic algorithm configuration framework. *Journal of Artificial Intelligence Research*, 36(1):267–306, 2009.
17. Darwin Klingman. NETGEN: A program for generating large scale (un)capacitated assignment, transportation, and minimum cost flow network problems. Technical, Texas University, Office of Naval Research, 1973.
18. Martha G. Pilcher and Ronald L. Rardin. Partial polyhedral description and generation of discrete optimization problems with known optima. *Naval Research Logistics (NRL)*, 39(6):839–858, 1992.
19. Soubhik Chakraborty and Pabitra Pal Choudhury. A statistical analysis of an algorithm's complexity. *Applied Mathematics Letters*, 13(5):121–126, 2000.
20. Carlos Cotta and Pablo Moscato. A mixed evolutionary-statistical analysis of an algorithm's complexity. *Applied Mathematics Letters*, 16(1):41–47, 2003.
21. Jano Van Hemert. Evolving Combinatorial Problem Instances That Are Difficult to Solve. *Evolutionary Computation*, 14(4):433–462, 2006.
22. Wanru Gao, Samadhi Nallaperuma, and Frank Neumann. Feature-Based Diversity Optimization for Problem Instance Classification. In *International Conference on Parallel Problem Solving from Nature*, pages 869–879. Springer, 2016.
23. Robert E. Bixby. A brief history of linear and mixed-integer programming computation. *Documenta Mathematica*, pages 107–121, 2012.
24. Elias Boutros Khalil, Pierre Le Bodic, Le Song, George L. Nemhauser, and Bistra N. Dilkina. Learning to Branch in Mixed Integer Programming. In *AAAI*, pages 724–731, 2016.
25. Giovanni Di Liberto, Serdar Kadioglu, Kevin Leo, and Yuri Malitsky. DASH: Dynamic Approach for Switching Heuristics. *European Journal of Operational Research*, 248(3):943–953, February 2016.

26. Kevin Leyton-Brown, Eugene Nudelman, and Yoav Shoham. Empirical hardness models: methodology and a case study on combinatorial auctions. *Journal of the ACM*, 56(4):1–52, 2009.
27. Serdar Kadioglu, Yuri Malitsky, Meinolf Sellmann, and Kevin Tierney. ISAC-Instance-Specific Algorithm Configuration. In *ECAI*, volume 215, pages 751–756, 2010.
28. Tobias Achterberg. SCIP: Solving constraint integer programs. *Mathematical Programming Computation*, 1(1):1–41, 2009.
29. Laura A Sanchis and Mark A Fulk. On the efficient generation of language instances. *SIAM Journal on Computing*, 19(2):281–296, 1990.
30. Simon Bowly. MIP test instance generation via constructed LP relaxations, April 2017. `https://doi.org/10.5281/zenodo.556009`.