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

Generation techniques for linear 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 programming. We propose a generation framework to shift instance generation and instance space search 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 parameterised generators and search algorithms using this framework generate only instances with this property.

Our results demonstrate that controlled 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 algorithm performance.

[*] Corresponding author.

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

S. Bowly · K. Smith-Miles
School of Mathematics and Statistics
Melbourne University, Parkville VIC 3010, Australia
E-mail: sbowly@student.unimelb.edu.au

K. Smith-Miles
E-mail: smith-miles@unimelb.edu.au

D. Baatar
School of Mathematical Sciences
Monash University, Clayton VIC 3800, Australia
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 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 [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 which 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.

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 complete 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 'naive' approach, which generates a distribution of random instances.

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]. Given the proliferation of portfolio-based and automatically configured solvers [13, 14], and the need for representative test instances in the tuning of such strategies [15, 16], 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 [17] 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 [18] 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 [19] 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. 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 [5]. 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 [20] and Cotta and Moscato [21] 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 [22] and diversity of measured features [5, 23]. 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 linear programming test instances with controllable properties. We present a framework which allows for comparison of a ‘naive’ random generator with a highly parameterised generator. The framework also incorporates methods for using 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 problem space, such as where phase transitions occur or where challenging instances have previously been found. This would not be possible using static tests sets or naive random generation methods, which provide limited feature control.

In future research we aim to progress to generator design for mixed integer programs, so in this paper we focus in particular on generating feasible, bounded LP instances. A clear requirement for generated MIP problems would be boundedness and feasibility of the root node LP relaxation. 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 same applies to node relaxations during branch and bound; there is no need to branch further if the node relaxation is found to be infeasible, so such instances may be of limited applicability. Furthermore, existing feature based analysis and performance prediction methods for MIP [13, 14] use features related to the LP relaxation solution, which are not relevant in infeasible or unbounded cases.

We focus 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 [24]. 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. [25] use integrality slack distances to select branching variables, and Liberto et al. [26] 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 remainder of the paper is organised as follows. In Section 2 we describe the characteristics of linear programming instances that we wish to control. A general framework for the design of parameterised generators to control properties of instances is presented in Section 3. The specific implementation of this framework to generate feasible and bounded linear programming instances is presented in detail in Section 4. Results are presented in Section 5 before conclusions are drawn in Section 6.

2 Linear Programming Instances

This section introduces notation, properties and measured features for linear programming instances. Given our aim to build towards a MIP generator, we consider features which 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. [25] use integrality slack distances to select branching variables, and Liberto et al. [26] use measures of active constraints to select heuristics. We are also interested in being able to control feasibility of generated instances. This refines our search space and allows the relationships between features and the feasibility phase transition to be explored. In the MIP context, restricting attention to feasible, bounded LP cases prevents production of trivial root node LP cases which are not likely to be informative of MIP solver performance.

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)
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 $\{\frac{a_{ji}}{b_j} \mid b_j \neq 0\}$ (min/mean/max)
Column-normalised coefficient statistics $\{\frac{a_{ji}}{c_i} \mid c_i \neq 0\}$ (min/mean/max)
Constraint degree-normalised rhs statistics $\{\frac{b_j}{d(v_j)}\}$ (min/mean/max)
Variable degree-normalised objective statistics $\{\frac{c_i}{d(u_i)}\}$ (min/mean/max)

Table 1: Linear programming features.

The LP generator produces instances in canonical form:

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

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

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

Any feasible bounded LP instance can alternatively be represented by a constraint matrix A and optimal values of the primal, slack, dual and surplus variables $(\hat{x}, \hat{s}, \hat{y}, \hat{r})$ for the above formulations. This specification is enough to uniquely determine the instance data (A, b, c).

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 linear programming instances. Features used in this work have been previously used for performance prediction in MIP [14], combinatorial auctions [27] and set covering [28]. We note that not all features in this table are uniquely defined. Statistical features of the constraint, rhs and objective coefficient features do not vary, however the relaxation solution features need not be unique if the problem has degenerate optima. This phenomenon is further explored in Section 4 where the instance construction method is defined.

3 A General Framework for Generator Design

In this section we present a general framework for parameterised generation and local search in a restricted instance space. We use the traveling salesman problem (TSP) as a running example to highlight the problem specific components which need to be developed each time the framework is applied to a new problem domain. The TSP is chosen as an illustrative example in this section for two reasons: firstly for its simplicity as a problem to enable the framework detail to be presented clearly; and secondly to show the broader applicability of the framework beyond the LP setting discussed in the subsequent sections.

A TSP instance is a weighted undirected graph where edge weights represent the distance between connected pairs of locations. An instance is feasible if its graph has at least one Hamiltonian cycle. In this section we demonstrate the application of the framework by showing the steps required to develop a generator for feasible TSP instances.

3.1 Problem Space

The general problem space Π is the set of all instances of the general problem. Let the problem space $\mathcal{P}$ be the subset of instances of Π over which we are interested in studying algorithm performance. Specifically, $\mathcal{P}$ is defined by the outcome of a decision problem on instances of the general problem, which splits Π into the set $\mathcal{P}$ and its complement $\bar{\mathcal{P}}$.

For the TSP example, we are interested in studying algorithm performance when solving feasible instances. Π is the set of all possible TSP instances. A TSP instance $x \in \Pi$ is represented as a weighted undirected graph with N vertices, or more concretely, a list of entries (i, j, w) each defining an edge between vertices i and j with positive weight w. $\mathcal{P}$ is the set of all feasible TSP instances. The obvious issue here is that determining whether an instance x is in $\mathcal{P}$ is a hard decision problem. The outcome of this decision problem is not clear from inspection of the problem data in this typical form.

3.2 Generator Requirements

A generator G_Π for the general problem space Π is an algorithm which produces instances at random in Π but cannot specify in advance whether a particular generated instance belongs to $\mathcal{P}$ or $\bar{\mathcal{P}}$. We refer to such a generator as *naive*. By contrast, a *controllable* generator $G_\mathcal{P}$ for the problem space $\mathcal{P}$ is an algorithm which produces instances at random which are guaranteed to lie in $\mathcal{P}$. $G_\mathcal{P}$ must be *correct* in that it produces instances only of $\mathcal{P}$, not of $\bar{\mathcal{P}}$, and *complete* in that it is capable of producing any instance in $\mathcal{P}$.

An example of a naive generator G_Π for general TSP instances would be an algorithm which chooses edges uniformly at random with probability p, and chooses weights from some distribution of strictly positive values. While p influences the likelihood that a given TSP instance has a Hamiltonian cycle, whether such a cycle exists cannot be specified as a strict condition. Defining a controllable generator $G_\mathcal{P}$, which guarantees that the resulting graph has a Hamiltonian cycle, and furthermore which can produce any possible graph with such a cycle, is clearly a more challenging task. Randomly generating weighted edges to build the typical representation of a TSP instance is unlikely to achieve this goal.

3.3 Encoding

To aid in developing a correct and complete generator for $\mathcal{P}$, we define an encoding E which admits only instances in $\mathcal{P}$. An encoded form $e \in E$ is an alternative representation of the problem data of instance x. Every valid encoded form must represent an instance of $\mathcal{P}$ (correctness), and every member of $\mathcal{P}$ must have a valid encoded form (completeness). As a consequence of this requirement, all instances in $\bar{\mathcal{P}}$ do not have a valid encoded form. The *constructor* $C_{\mathcal{P}}$ is an algorithm which builds the original problem data representation from the encoded form, so that $x = C_{\mathcal{P}}(e)$. A generator G_E for the encoding E produces encoded forms e at random. It must be possible for any encoded form $e \in E$ to be produced by this generator.

One possible encoded form E for TSP instances with a feasible tour is the following:

- an ordering of the N vertices (defining a Hamiltonian cycle),
- a list of N weights on the edges in this cycle, and
- a list of additional weighted edges.

The constructor produces a typical TSP representation (list of edges) by creating N edges from the cycle ordering and weights and taking the union with the list of additional edges.

It is easy to show that this encoding is correct and complete. The first step of the constructor always includes a feasible tour in the resulting instance. Furthermore, a Hamiltonian cycle can be extracted from any TSP instance with a feasible tour, which defines the ordering of N vertices in an encoded form of the instance. Since every encoded form produces a valid instance of $\mathcal{P}$, and a valid encoded form can be defined for every instance of $\mathcal{P}$, this encoding is correct and complete.

A random generator G_E for encoded form TSP instances would, for a given problem size N, produce a random ordering of the vertices $v_1 \dots v_N$ and a random set of additional edges. The first phase of the algorithm is just a random shuffle of $1 \dots N$ to define an ordering. The second phase generates weights for the cycle edges from a non-negative distribution. The final phase can proceed to choose weighted edges with probability p as with the generator for the general problem. This algorithm is clearly correct and complete with respect to the encoding E.

Note that there is redundancy in this encoding. Edges of the generated graph can be defined either as part of the cycle definition or as one of the additional edges. This means that an instance with multiple Hamiltonian cycles may be represented by multiple valid encoded forms, possibly resulting in bias towards such instances in the controllable generator. We note that although it may be possible to avoid this issue, encoding redundancy does not impact the definitions of correctness and completeness.

Defining an encoding splits the instance generation task into a two stage process: G_E and $C_{\mathcal{P}}$. By showing that both stages are correct and complete, the resulting generator $G_{\mathcal{P}}$ is clearly correct and complete with respect to the target problem space $\mathcal{P}$.

3.4 Neighbourhood Operators

While a correct and complete generator is theoretically capable of producing any instance in the defined search space, we cannot infer from this that all instance classes are equally likely to occur. In particular, it may not always be possible to control the values of specific features, or the difficulty of an instance for a particular algorithm. As a result a generated test set will likely not be balanced across all feature ranges. Some feature values may be missing entirely as they appear with a very low probability. Local search may be advantageous in addressing such deficiencies.

Developing a local search algorithm requires that we define neighbourhood operators for the problem space. As with the initial generator design, manipulating instances in $\mathcal{P}$ in their typical form does not guarantee that the resulting neighbour is also in $\mathcal{P}$. Instead we implement neighbourhood operators in the encoding space. A general form for these operators can be defined by replacing part of an encoded form with new data produced by the generator G_E. The resulting encoded form is passed to the constructor to produce a neighbouring instance.

To illustrate this, first note that given a TSP instance, we can produce a new instance simply by adding and removing edges with a given probability. However, operating directly on the graph representation of the instance could break an existing Hamiltonian cycle, resulting in a neighbour which is in $\bar{\mathcal{P}}$. Avoiding this requires a repair heuristic in order to keep this neighbour in the space $\mathcal{P}$. While this would guarantee membership of $\mathcal{P}$, it would be hard to verify that the search space is entirely connected by such an operator, and therefore whether a target instance can be reached over many steps of a local search algorithm.

Instead, we define a neighbourhood operator in the encoding space which uses G_E to generate new data for the instance. For TSP, this operator would alter the encoded form by:

1. reshuffling part of the cycle ordering,
2. generating new weights for some cycle elements, and
3. adding or removing additional edges with a given probability.

A new TSP instance is produced by construction from this encoded form. Given that G_E is complete and correct, it is easy to verify that a finite number of applications of this operator can progress between any pair of encoded forms. This will be demonstrated later in the LP case study.

3.5 Search Objectives

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 diversity. A search algorithm, using neighbourhood operators as explained above, optimises these sufficiency metrics by generating new instances. For an instance set which does not cover the full feature range the objective should favour instances which increase feature diversity. 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.

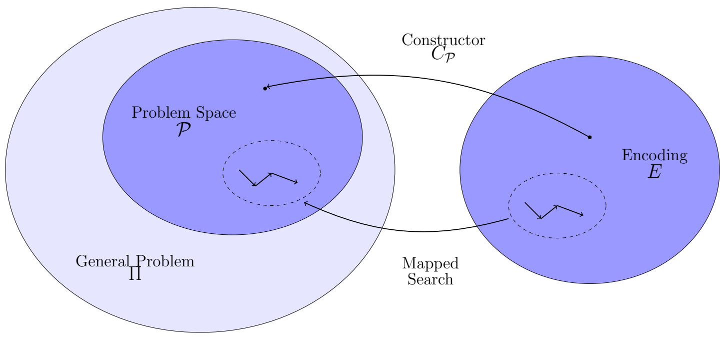


Fig. 1: The constructor shifts an instance generation and search problem to a different space by means of an encoded representation of instance data.

To formalise these search objectives, we define functions which produce a diversity metric from an instance set. We then define $\Gamma(I)$ and $\Lambda(I)$ which map an instance set $I \subset \mathcal{P}$ to a diversity measure for feature values or performance metrics, respectively. For example, given two functions f_1 and f_2 which calculate features for an instance x, and target values g_1 and g_2 for those features, we may define

$$\Gamma(I) = min\{\, (f_1(x) - g_1)^2 + (f_2(x) - g_2)^2 \mid x \in I \subset \mathcal{P}\}.$$

This diversity measure calculates the distance to the target point of the closest instance in the set. A local search algorithm minimises this metric to produce new instances, continuing until a given threshold of sufficiency is met.

In performance space, local search aims to generate harder instances. One example for the TSP problem is to maximise the number of 2-opt iterations required to generate a good heuristic solution for a given instance. We may define the function

$$\Lambda(I) = max\{\, \text{2-opt}(x) \mid x \in I \subset \mathcal{P}\}$$

to be maximise in local search in order to achieve this goal. More complex functions may be defined which consider, for example, the difference in performance between two algorithms, or the variance of values in feature space. In our investigations of local search in instance space, we consider the metrics defined here: minimisation of distance to a target point in feature space and maximisation of difficulty for a target algorithm.

3.6 Summary

We are given the task of implementing a generator $G_{\mathcal{P}}$ for instances $x \in \mathcal{P} \subset \Pi$ which is correct and complete with respect to $\mathcal{P}$. Designing such a generator, and proving that it is correct and complete, is simplified by shifting the instance generation process to an alternative encoding of instance data.

Figure 1 summarises this process: we define the constructor $C_{\mathcal{P}}$ such that the image of the encoding E is the target problem space $\mathcal{P} \subset \Pi$. Instance generation and local search in this encoded space is then unrestricted. Figure 2 gives a process

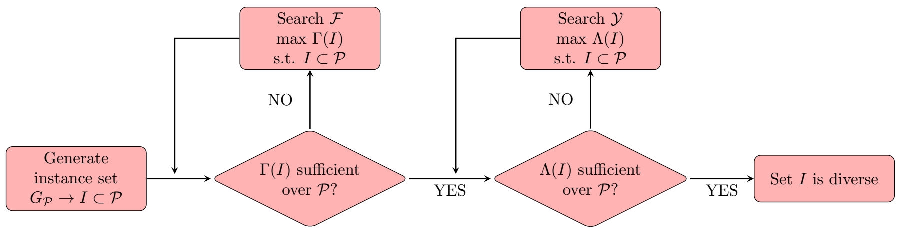

Search $\mathcal{F}$
max $\Gamma(I)$
s.t. $I \subset \mathcal{P}$
Search $\mathcal{Y}$
max $\Lambda(I)$
s.t. $I \subset \mathcal{P}$
NO
NO
Generate instance set $G_{\mathcal{P}} \rightarrow I \subset \mathcal{P}$
$\Gamma(I)$ sufficient over $\mathcal{P}$?
YES
$\Lambda(I)$ sufficient over $\mathcal{P}$?
YES
Set I is diverse

Fig. 2: The initial generated instance set is updated with new instances produced by instance space search. This search may be performed in feature space $\mathcal{F}$ or algorithm performance space $\mathcal{Y}$.

for updating the instance set with new instances found by searching feature and performance space until the sufficiency requirements are met.

Application of this framework to a new problem space requires the user to develop:

- an encoding admitting only instances of the target problem space,
- a construction algorithm to produce instances from the encoded forms, and
- a generator for encoded forms.

The focus of this paper is linear programming. We used the TSP as a running example in this section to show the implementation of the framework, since the components for that domain are very simple to describe. In the following section we show how we construct LP instances from an encoded form to guarantee that the resulting problems are feasible and bounded, and develop a generator for this encoding which controls the features of interest.

4 Linear Programming Implementation

In this section we implement the domain specific components of the instance generation framework for linear programming. The general problem space Π is the set of all linear programming instances, and $\mathcal{P}$ is the set of all feasible, bounded linear programs. Instances are represented as a tuple (A, b, c) which defines the canonical form of the primal problem.

Determining whether a given instance x is in the target set $\mathcal{P}$ requires an algorithm to solve the corresponding linear programs. If such an algorithm terminates at an optimal point, then $x \in \mathcal{P}$. If the result is a proof of infeasibility or unboundedness, then $x \in \bar{\mathcal{P}}$. The result of this decision problem cannot otherwise be determined from the problem data in its typical form.

In Section 4.1 we define a naive generator G_Π for general LP instances, which does not guarantee that instances are feasible and bounded. In Section 4.2 we define an encoding E 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 Section 4.3 we define a generator G_E for encoded forms $e \in E$ of feasible, bounded linear programs, and show that it is correct and complete. Section 4.4 gives neighbourhood operators which are developed using the same random processes as each generator.

For both the naive 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 naive generator cannot be used to search the space $\mathcal{P}$ without the use of inefficient repair operations.

4.1 Naive Generator

The naive 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 parameterises the corresponding features.

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. 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 producing empty constraints or unbounded variables. Nonzero values in the constraint matrix are then drawn from a normal distribution.

4.2 Encoding and Construction

The naive generation routine described in Section 4.1 is clearly not sufficient to define a controllable generator $G_{\mathcal{P}}$ which is complete and correct. Instead we define an encoding for linear programming 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, first consider the key characteristic of instances $x \in \mathcal{P}$. The primal problem (P) and dual problem (D) must both be

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 $A \in \boldsymbol{Q}^{m\times n}$.
 1: Set $d(u_i) = 1$ for randomly selected $i$, 0 for all others
 2: Set $d(v_j) = 1$ for randomly selected $j$, 0 for all others
 3: $e \leftarrow 1$
 4: while $e < \rho mn$ do
 5:     $s \leftarrow$ draw $n$ values from $U(0,1)$
 6:     $t \leftarrow$ draw $m$ values from $U(0,1)$
 7:     Increment the degree of variable node $i$ with maximum $p_v \frac{d(u_i)}{e} + s_i$
 8:     Increment the degree of constraint node $j$ with maximum $p_c \frac{d(v_j)}{e} + t_j$
 9:     $e \leftarrow e + 1$
10: end while
11: for $i = 1 \dots n$ do
12:     for $j = 1 \dots m$ do
13:         $r \leftarrow$ draw from $U(0,1)$
14:         if $r < \frac{d(u_i)d(v_j)}{e}$ then
15:             Add edge $(i,j)$ to $VC$
16:         end if
17:     end for
18: end for
19: while $\min(d(u_i), d(v_j)) = 0$ do
20:     Choose $i$ from variable vertices with degree 0, or randomly if all $d(u_i) > 0$
21:     Choose $j$ from constraint vertices with degree 0, or randomly if all $d(v_j) > 0$
22:     Add edge $(i,j)$ to $VC$
23: end while
24: for $(i,j) \in E(VC)$ do
25:     $a_{ji} := N(\mu_A, \sigma_A)$
26: end for
27: return A
```

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 basic feasible solutions to the primal and dual. 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 basic feasible solution is encoded using a basis B for the primal problem and its complement N for the dual. The constructor $C_{\mathcal{P}}$ enforces that only those primal problem variables in B, and dual problem variables in N are non-zero, ensuring the complementary slackness conditions are always satisfied. Ensuring feasibility of the given solution is then the only requirement to design an instance for which these solutions are optimal.

The encoding E represents instances as tuples (n, m, A, α, β). The domain of the encoding is all such tuples 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 $C_{\mathcal{P}}$ will return a feasible, bounded linear program with n variables and m constraints in canonical form (A, b, c). Algorithm 2 gives the constructor algorithm.

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 $i = 1 \dots n$ do
 2:     $\hat{x}_i \leftarrow \beta_i \alpha_i$
 3:     $\hat{r}_i \leftarrow (1 - \beta_i)\alpha_i$
 4: end for
 5: for $j = 1 \dots m$ do
 6:     $\hat{y}_j \leftarrow (1 - \beta_{j+n})\alpha_{j+n}$
 7:     $\hat{s}_j \leftarrow \beta_{j+n}\alpha_{j+n}$
 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 *Outputs of the constructor are feasible, bounded linear programs. Furthermore, if the encoded solutions correspond to non-degenerate bases of A, then $(\hat{x}, \hat{s})$ is an optimal basic feasible solution for the primal and $(\hat{y}, \hat{r})$ is an optimal basic feasible solution for the dual.*

Proof Consider the standard form of the primal problem (P) and the corresponding standard form of the dual problem (D). 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 2. 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.

If the corresponding submatrices of A for the solution bases are non-singular, then $(\hat{x}, \hat{s})$ is a basic feasible solution to (P) since $\sum \beta_i = m$ and $(\hat{x}, \hat{s})$ are nonzero only for $\beta_i = 1$. Similarly, $(\hat{y}, \hat{r})$ is a basic feasible solution to (D) since $\sum 1 - \beta_i = n$ and $(\hat{y}, \hat{r})$ are nonzero only for $\beta_i = 0$. Furthermore, $(\hat{x}, \hat{s})$ and $(\hat{y}, \hat{r})$ satisfy the complementary slackness conditions, since

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

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

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 encoded form $e \in E$.*

Proof Any feasible, bounded LP instance in standard form has at least one optimal basic feasible solution, defined by the basis B containing $m + n$ 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 $\beta_i = 1$ 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 $\sum \beta_i = m$. The matrix A is not changed in the encoded form. Therefore $e = (n, m, A, \alpha, \beta)$ defines an encoded linear program in E, for which $x = C_{\mathcal{P}}(e)$. Therefore any feasible, bounded linear program P has a corresponding encoding $e \in E$ and is produced by the constructor given appropriate inputs satisfying Equation (1).

Proposition 1 implies that the encoding is correct, since Algorithm 2 will produce a feasible, bounded linear program for all valid encoded forms. Proposition 2 implies that the encoding is complete since for any feasible bounded linear program x there exists an encoded form e for which $x = C_{\mathcal{P}}(e)$. Following the methodology in Section 3, we now only need to define a correct and complete generator G_E for encoded forms. Clearly, it is significantly easier to develop a generator for encoded forms satisfying Equation (1) than to attempt to extend the naive 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). Any instance with multiple optimal bases for the primal or dual problem will have multiple encodings. This can occur when 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 random encoded strings may favour instances with multiple representations.

4.3 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 Section 4.1, this process controls features of the constraint matrix.

The second stage generates the optimal solution for the constructed instance, including primal and dual solution values, primal constraint slack values and dual constraint surplus values. 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 3 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.

Algorithm 3 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 $(\alpha,\beta)$
 1: $k \leftarrow [\gamma \min(n,m)]$
 2: $P_{opt} \leftarrow$ Choose $k$ indices from $\{1 \dots n\}$ without replacement.
 3: $S_{opt} \leftarrow$ Choose $m-k$ indices from $\{1 \dots m\}$ without replacement.
 4: $P_{frac} \leftarrow$ Choose $[\lambda n]$ indices from $\{1 \dots n\}$ without replacement.
 5: for $i = 1 \dots n$ do
 6:     $X_1 \leftarrow$ draw from LogNormal$(\mu_s, \sigma_s)$
 7:     $X_2 \leftarrow$ draw from Beta$(a, a)$,
 8:     if $i \in P_{opt}$ then $\beta_i := 1$ else $\beta_i := 0$ end if
 9:     if $i \in P_{frac}$ then $\alpha_i := \lceil X_1 \rceil - X_2$ else $\alpha_i := \lceil X_1 \rceil$ end if
10: end for
11: for $j = 1 \dots m$ do
12:     $X_3 \leftarrow$ draw from LogNormal$(\mu_s, \sigma_s)$.
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 $(\alpha,\beta)$
```

Given the choice of primal and slack variable indices in Algorithm 3, 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 Equation (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 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 $\alpha_i > 0$. Therefore $\alpha \geq 0$ as required by Equation (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, $|P_{opt}| = [\gamma \min(n,m)]$ and $|S_{opt}| = m - [\gamma \min(n,m)]$. 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 $|P_{frac}| = [\lambda \, |P_{opt}|]$.

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 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:

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

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.

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.

4.4 Instance Space Search

The generators defined above can be used to define neighbourhood operators which replace part of the instance data. First consider the naive 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 naive 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.

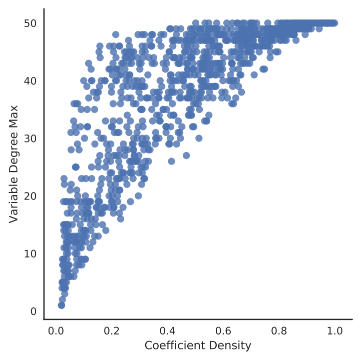
Alternatively, neighbourhood operators can be defined in the encoding space. Given an encoded form with m rows and n columns, generate a new instance using $G_{\mathcal{P}}$ with one row and n columns. This generates a new row of A as above, and corresponding values α_i and β_i. A new encoded form is 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 first 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.

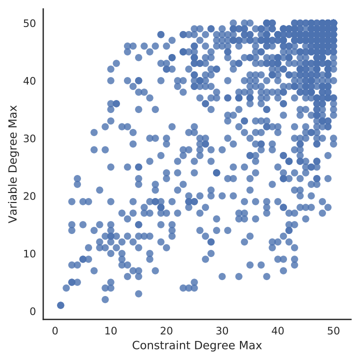
5 Experimental Results

This section compares the diversity of instances produced by the naive and controllable generations in feature and performance space. The naive 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 these gaps. Additionally, we use search in performance space to produce instances which are harder for several LP algorithms.

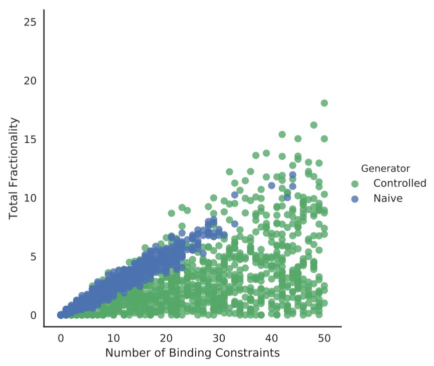
Generation and search is implemented in Python 3.6.3, with a C++ extension using the Coin LP callable library to facilitate feature computation. The pseudo-random number generator included with the numpy library (version 1.14.2) for python is used for all random number generation. Generation, feature 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



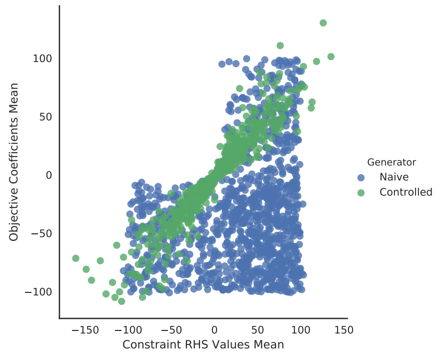
(a)



(b)



(c)



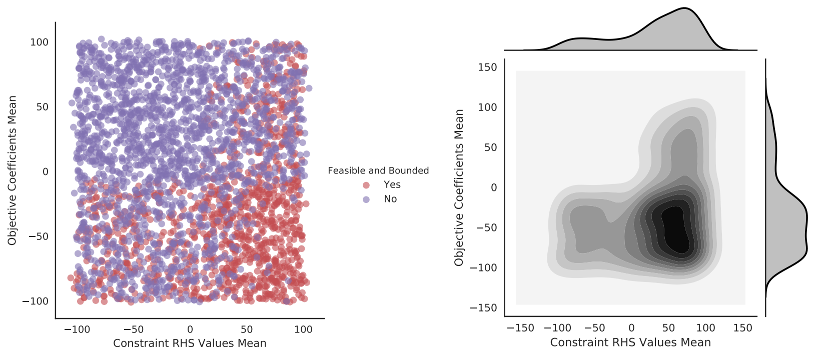
(d)

Fig. 3: Distribution of instances in feature space. Figures 3a and 3b show features of the constraint matrix, which are common to the two generators. Figures 3c and 3d compare distributions of features where the generators differ, showing only feasible bounded instances.

simplex and barrier solvers on generated instances. 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 from the Zenodo repository at doi:10.5281/zenodo.556009 [29].

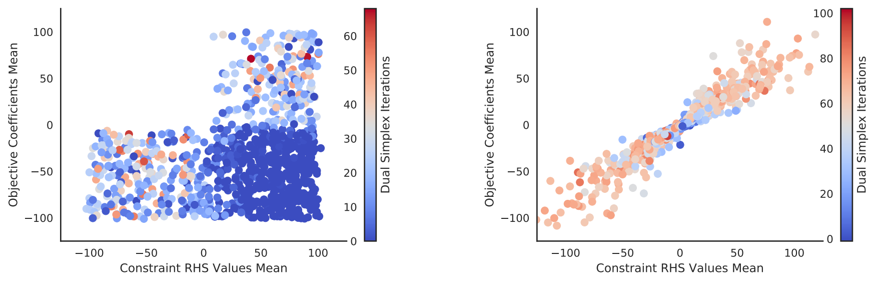
5.1 Parameterised Generation

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



(a) (b)

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

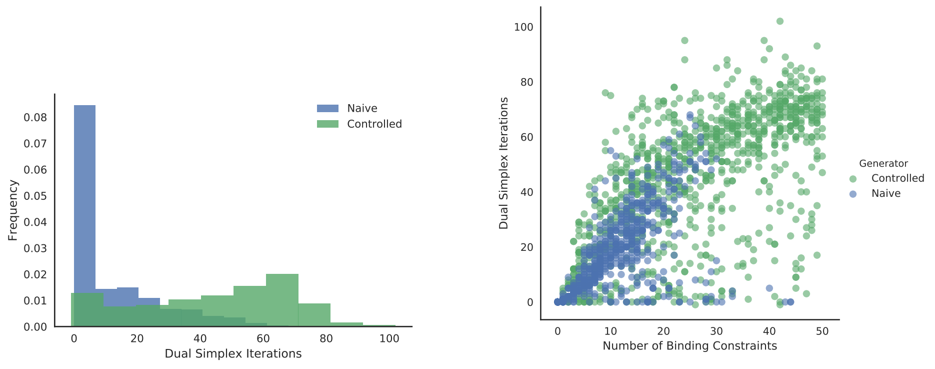


(a) (b)

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. Figure 5a shows instances produced using the naive generator. Figure 5b shows instances produced using the controllable generator.

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. Size-independent parameter distributions used to vary instance features are given in Table 2. Note that these parameter ranges are an arbitrary choice to demonstrate the 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



(a) (b)

Fig. 6: Comparison of the two generators in terms of instance difficulty for the dual simplex algorithm. Figure 6a shows frequency distributions for the two sets. Figure 6b shows a possible explanatory variable which is controlled for in the controllable generator but not the naive generator.

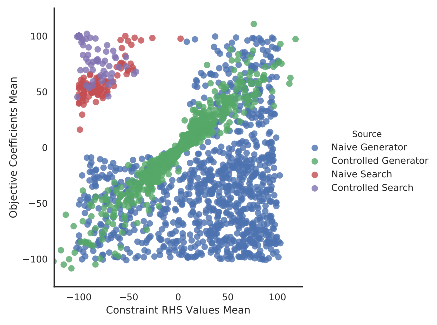
Parameter	Value/Distribution
Variables n	50
Constraints m	50
Density ρ	$U(0.1, 1)$
High degree variables p_v	$U(0, 1)$
High degree constraints p_c	$U(0, 1)$
Coefficient Mean μ_A	$U(-2, 2)$
Coefficient StDev σ_A	$U(0.1, 10)$
Primal vs slack basis γ	$U(0, 1)$
Fractional primals λ	$U(0.1, 1.0)$
Beta fractionality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

Table 2: Generator parameters used in experiment.

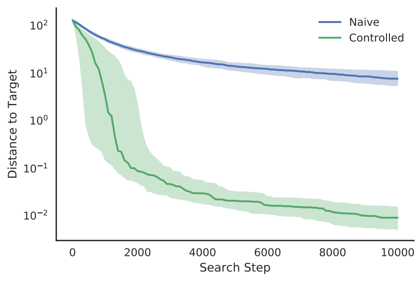
and 3000 instances from the naive generator. The larger sample size is required to produce an equal number of feasible bounded instances using the naive generator since only 37.5% of instances generated using the naive method satisfy this property.

Figures 3a and 3b show features of the constraint matrix, where each scatter plot point represents the feature values of a generated instance. The naive 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. Contrasting this with a uniform random approach, which would produce approximately uniform degree in most cases, the generator is able to improve feature diversity in this space.

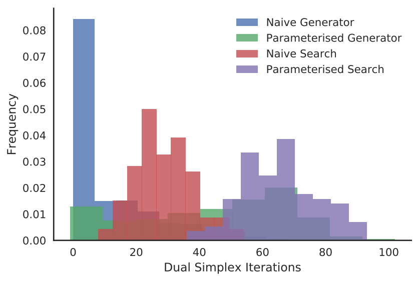
Figure 3c shows the joint distribution of the number of binding constraints at the optimal point and total fractionality of the primal solution values. This demon-



(a)



(b)



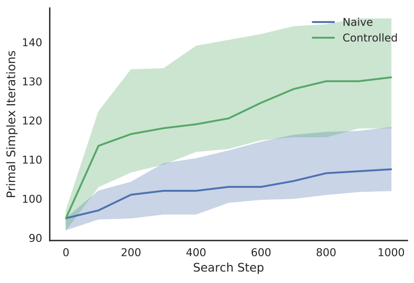
(c)

Fig. 7: Local search results in feature space. Figure 7a shows original generated instances and results of local search after 1000 steps. Figure 7b shows progress made by the median result with upper and lower quartiles. Figure 7c compares instance hardness for the result sets.

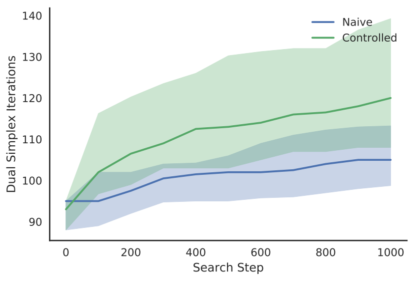
strates the key additional parameters available to the controllable generator; it can explicitly set the primal, dual, slack and reduced cost values. By contrast, the naive generator occupies a very narrow band in this feature space. More importantly, the parameters of this generator have no bearing on these features, so it is not possible to alter the input parameter distributions in order to achieve more variation.

There are two other observations to be made here. First, we do 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 of course 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.

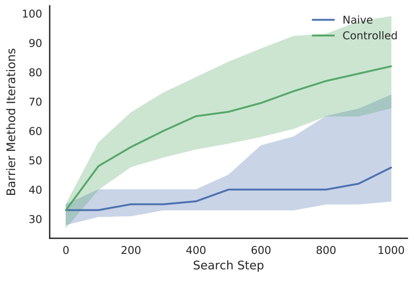
Secondly, the effects of solution degeneracy result in a mismatch between our expected feature distribution (which we can calculate using formulae derived in Section 4.3) and the actual distribution. While the constructor guarantees that the solution specified in the encoded form is an optimal basic feasible solution, it does



(a)



(b)



(c)

Fig. 8: Progression of local search runs which aim to generate harder instances.

not guarantee that it is unique. When we calculate instance features by solving the LP from scratch, we may arrive at a different optimal point, and therefore measure a different feature value from the design value. This is an interesting avenue for further investigation, particularly in MIP where learning algorithms use this feature in algorithm parameter tuning.

Figure 3d 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 naive generator achieves more diversity in this feature space; it produces instances in the second quadrant while the controlled generator is more restrictive. The controlled 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 $r_i, s_j >= 0$, it is easy to see that we could drive instances towards the lower right corner of Figure 3d 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 4a shows the original sample points of the naive generator, with both feasible bounded and infeasible or unbounded instances shown. The set of *all* naively generated instances is independently uniformly distributed in this two dimensional space. As previously mentioned, 37.5% of instances from the naive 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 Figure 4b as a probability density function. We remind the reader that this phase transition does not apply to all linear programs; it is specific to the parameter distribution and 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 solution $\hat{x} = \bar{0}$ is feasible to the primal program P. Similarly, if $c_i < 0$ for all variable indices i, then the solution $\hat{y} = \bar{0}$ 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 Figure 4a to be more likely to be feasible and bounded. However we further note that such instances are likely to be uninteresting, as the $\bar{0}$ solution is likely to be optimal, resulting in a trivially 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 Figure 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 (see Figure 5b), which produces instances almost exclusively in this region, results in harder to solve instances.

The disparity in difficulty between the instances produced by the two generators may also be explained by the controllable generators ability to control the number of binding constraints. Figure 6b 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 naive 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 a target algorithm.

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

5.2 Instance Space Search

Our search efforts in feature space focus on the missing region in Figure 3d. The generators used in Section 5.1 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 naive 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 10000 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 naive operator. This is likely due to the high rejection rate of neighbours in the naive process. 40-50% of neighbouring instances produced using the naive operator are infeasible or unbounded and thus immediately rejected.

Figure 7c shows that although the new instances produced by feature space search are not trivially solvable, no new instances were found which were harder to solve than those in the original generated data sets. The next step, then, is to apply search to performance space. The algorithm proceeds in the same manner as feature search, accepting new instances which increase the difficulty metric for the target algorithm.

Figure 8 shows search progression using the naive 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. Each run of local search is terminated after 1000 steps. Numerical results are given in Table 4. Again, the controllable method is in general more efficient at producing harder instances in a given number of steps, however the performance difference is not as pronounced as feature space search.

The results of performance space search demonstrate the benefits of local search in instance space particularly well. The initial test set produced by the controllable generator results in, at best, instances which take 102 instances to solve with the primal simplex algorithm. Local search, which maximises difficulty for the primal simplex algorithm, produces instances which take nearly twice as many iterations to solve. 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.

6 Conclusion

This paper proposes an instance generation framework which is applied to produce linear programming instances with controllable properties. The framework aids in

		Distance to Target Point	
		median	min
Method	Step		
Controllable Search	0	129.686	107.561
	100	96.957	0.380
	1000	3.479	0.022
	10000	0.009	0.000
Naive Search	0	129.476	107.561
	100	118.289	96.888
	1000	52.075	36.016
	10000	7.688	0.001

Table 3: Feature space search results. Shows the distance to target point achieved by local search at a given step. Results are aggregated over 100 trials for each neighbourhood operator.

		Primal Simplex		Dual Simplex		Barrier Method	
		max	median	max	median	max	median
Method	Step						
Controllable Generator	-	102	51	102	48	40	15
Controllable Search	0	102	95	102	93	40	33
	200	188	116	148	106	281	54
	500	188	120	175	113	505	66
	1000	188	131	190	120	505	82
Naive Generator	-	85	4	68	2	51	9
Naive Search	0	102	95	102	95	40	33
	200	134	101	131	97	138	35
	500	154	103	168	102	138	40
	1000	190	107	172	105	140.0	47

Table 4: Performance space search results. Shows the number of iterations required by the target algorithm to solve the current instance at a given step. Results are aggregated over 100 trials for each neighbourhood operator and target algorithm. Comparison statistics for the generated instances (without search) is also shown.

the design of generators which produce a targeted subclass of a general problem. Using this framework, we have designed a generator for feasible, bounded LPs. We have demonstrated that the generator is theoretically capable of producing any feasible bounded LP, given the right parameter choices. Local search can then be applied to generate instances with characteristics which are rarely produced by the generator, or are more difficult 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.

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 objectives considered here, it is likely that generating instances with more complex characteristics will require specially designed global search algorithms. Developing operators for such algorithms, and proving they connect the search space, will be significantly easier using the framework we have developed. The alternative encoding for instance data used in this framework 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 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 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 alternative parameters 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.

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. Michael J. Todd. Probabilistic models for linear programming. *Mathematics of Operations Research*, 16(4):671–693, November 1991.
19. 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.
20. Soubhik Chakraborty and Pabitra Pal Choudhury. A statistical analysis of an algorithm's complexity. *Applied Mathematics Letters*, 13(5):121–126, 2000.
21. Carlos Cotta and Pablo Moscato. A mixed evolutionary-statistical analysis of an algorithm's complexity. *Applied Mathematics Letters*, 16(1):41–47, 2003.
22. Jano Van Hemert. Evolving Combinatorial Problem Instances That Are Difficult to Solve. *Evolutionary Computation*, 14(4):433–462, 2006.
23. 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.

24. Robert E. Bixby. A brief history of linear and mixed-integer programming computation. *Documenta Mathematica*, pages 107–121, 2012.
25. 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.
26. 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.
27. 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.
28. Serdar Kadioglu, Yuri Malitsky, Meinolf Sellmann, and Kevin Tierney. ISAC-Instance-Specific Algorithm Configuration. In *ECAI*, volume 215, pages 751–756, 2010.
29. Simon Bowly. MIP test instance generation via constructed LP relaxations, April 2017. `https://doi.org/10.5281/zenodo.556009`.