The Cutting-Stock Approach to Bin Packing: Theory and Experiments

David L. Applegate * Luciana S. Buriol † Bernard L. Dillard ‡
David S. Johnson § Peter W. Shor ¶

Abstract

We report on an experimental study of the Gilmore-Gomory cutting-stock heuristic and related LP-based approaches to bin packing, as applied to instances generated according to discrete distributions. No polynomial running time bound is known to hold for the Gilmore-Gomory approach, and empirical operation counts suggest that no straightforward implementation can have average running time $O(m^3)$, where m is the number of distinct item sizes. Our experiments suggest that by using dynamic programming to solve the unbounded knapsack problems that arise in this approach, we can robustly obtain average running times that are $o(m^4)$ and feasible for m well in excess of 1,000. This makes a variant on the previously un-implemented asymptotic approximation scheme of Fernandez de la Vega and Lueker practical for arbitrarily large values of m and quite small values of ϵ.

We also observed two interesting anomalies in our experimental results: (1) running time *decreasing* as the number n of items increases and (2) solution quality improving as running time is reduced and an approximation guarantee is weakened. We provide explanations for these phenomena and characterize the situations in which they occur.

1 Introduction

In the classical one-dimensional bin packing problem, we are given a list $L = (a_1, ..., a_n)$ of items, a bin capacity B, and a size $s(a_i) \in (0, B]$ for each item in the list. We wish to pack the items into a minimum number of bins of capacity B, i.e., to partition the items into a minimum number of subsets such that the sum of the sizes of the items in each subset is B or less. This problem is NP-hard and has a long history of serving as a test bed for the study of new algorithmic techniques and forms of analysis.

Much recent analysis (e.g., see [3, 4, 6]) has concerned the average case behavior of heuristics under *discrete distributions*. A discrete distribution F consists of a bin size $B \in \mathbb{Z}^+$, a sequence of positive integral sizes $s_1 < s_2 < \cdots < s_m \leq B$, and an associated vector $\bar{p}_F = \langle p_1, p_2, \ldots, p_m \rangle$ of rational probabilities such that $\sum_{j=1}^{m} p_j = 1$. In a list generated according to this distribution, the ith item a_i has size $s(a_i) = s_j$ with probability p_j, chosen independently for each $i \geq 1$. The above papers analyzed the asymptotic expected performance under such distributions for such classical bin packing heuristics as Best and First Fit (BF and FF), Best and First Fit Decreasing (BFD and FFD), and the new Sum-of-Squares heuristic of [6, 7].

Three of the above algorithms are online algorithms, and for these the order of the items in the list is significant. However, if we are allowed to do our packing offline, i.e., with foreknowledge of the entire list of items to be packed, then there is a much more compact representation for an instance generated according to a discrete distribution: simply give a list of pairs (s_i, n_i), $1 \leq i \leq m$, where n_i is the number of items of size s_i. This is the way instances are represented in a well-known special case of bin packing, the one-dimensional *cutting-stock* problem, which has many industrial applications. For such problems, an approach using linear programming plus knapsack-based column generation, due to Gilmore and Gomory [14, 15], has for 40 years been the practitioner's method of choice because of its great effectiveness when m is small. The packings it produces cannot use more than $OPT(L)+m$ bins (and typically use significantly fewer) and although the worst-case time bound for the original algorithm may well be exponential in m, in practice running time does not seem to be a problem.

In this paper we examine the Gilmore-Gomory approach and some of its variants from an experimental point of view, in the context of instances generated according to discrete distributions. We do this both to

*AT&T Labs, Room C224, 180 Park Avenue, Florham Park, NJ 07932, USA. Email: david@research.att.com.

†UNICAMP – Universidade Estadual de Campinas, DENSIS/FEEC, Rua Albert Einstein 400 - Caixa Postal 6101, Campinas - SP - Brazil. Email: buriol@densis.fee.unicamp.br. Work done while visiting AT&T Labs.

‡Department of Mathematics, University of Maryland, College Park, MD 20742. Email: bld@math.umd.edu. Work done while visiting AT&T Labs.

§AT&T Labs, Room C239, 180 Park Avenue, Florham Park, NJ 07932, USA. Email: dsj@research.att.com.

¶AT&T Labs, Room C237, 180 Park Avenue, Florham Park, NJ 07932, USA. Email: shor@research.att.com.

get a clearer idea of how the Gilmore-Gomory approach scales as m, n, and B grow (and how to adapt it to such situations), and also to gain perspective on the existing results for classical bin packing algorithms. Previous experiments with Gilmore-Gomory appeared primarily in the Operations Research literature and typically concentrated on instances with $m \leq 100$, where the approach is known to work quite quickly [10, 11, 21]. The restriction of past studies to small m has two explanations: (1) most real-world cutting stock applications have $m \leq 100$ and (2) for m this small, true optimization becomes possible via branch-and-bound, with Gilmore-Gomory providing both lower and, with rounding, upper bounds, e.g., see [10, 11]). Here we are interested in the value of the LP-based approaches as approximation algorithms and hence our main results go well beyond previous studies. We consider instances with m as large as is computationally feasible (which in certain cases can mean $m = 50,000$ or more). This will enable us to pose plausible hypotheses about how running time and solution quality typically scale with instance parameters.

In Section 2 we describe the original Gilmore-Gomory approach and survey the relevant literature. In Section 3 we describe an alternative linear programming formulation for computing the Gilmore-Gomory bound using a flow-based model, independently proposed by Valério de Carvalho [8, 9] and Csirik et al. [6, 7]. This approach can be implemented to run in time polynomial in m, $\log n$, and B (a better bound than we have for Gilmore-Gomory) but to our knowledge has not previously been studied computationally. In Section 4 we discuss the key grouping technique introduced in the asymptotic fully-polynomial-time approximation scheme for bin packing of Fernandez de la Vega and Lueker [12], another algorithmic idea that does not appear to have previously been tested experimentally (and indeed could not have been tested without an efficient Gilmore-Gomory implementation or its equivalent). In Section 5 we describe the instance classes covered by our experiments and summarize what is known theoretically about the average case performance of classical bin packing heuristics for them. Our results and conclusions are presented in Sections 6.

2 The Gilmore-Gomory Approach

The Gilmore-Gomory approach is based on the following integer programming formulation of the cutting stock problem. Suppose our instance is represented by the list L of size/quantity pairs (s_i, n_i), $1 \leq i \leq m$. A nonnegative integer vector $\bar{p} = (p[1], p[2], \ldots, p[m])$ is said to be a *packing pattern* if $\sum_{i=1}^{m} p[i]s_i \leq B$. Suppose there are t distinct packing patterns $p_1, \ldots, p_t$ for the given set of item sizes. The integer program has a variable x_j for each pattern p_j, intended to represent the number of times that pattern is used, and asks us to minimize $\sum_{j=1}^{t} x_j$ subject to the constraints

$$\sum_{j=1}^{t} p_j[i]x_j = n_i, \quad 1 \leq i \leq m \tag{2.1}$$

$$x_j \geq 0, \quad 1 \leq j \leq t \tag{2.2}$$

The solution value for the linear programming relaxation of this integer program, call it $LP(L)$, is a lower bound on the optimal number of bins. Moreover, it is a very good one. For note that in a basic optimal solution there will be at most m non-zero variables, and hence at most m fractional variables. If one rounds each of these up to the nearest integer, one gets a packing of a superset of the items in the original instance that uses fewer than $LP(L) + m$ bins. Thus an optimal packing of the original set of items can use no more bins and so $OPT(L) < LP(L) + m$ in the worst case, and we can get a packing at least this good. In practice, a better rounding procedure is the following one, recommended by Wäscher and Gau [21]: Round the fractional variables *down* and handle the unpacked items using FFD. It is easy to prove that this "round down" approach also satisfies an $OPT(L) + m$ worst-case bound. Our experiments suggest that in practice its excess over $OPT(L)$ is typically no more than 4% of m.

There is an apparent drawback to using the LP formulation, however: the number t of packing patterns can be exponential in m. The approach suggested by Gilmore and Gomory was to avoid listing all the patterns, and instead generate new patterns only when needed. Suppose one finds a basic optimal solution to the above LP restricted to some particular subset of the patterns. (In practice, a good starting set consists of the patterns induced by an FFD packing of the items.) Let y_i, $1 \leq i \leq m$, be the dual variables associated with the solution. Then it is an easy observation that the current solution can only be improved if there is a packing pattern p' not in our subset such that $\sum_{i=1}^{m} p'[i]y_i > 1$, in which case adding the variable for such a pattern may improve the solution. (If no such pattern exists, our current solution is optimal.) In practice it pays to choose the pattern with the *largest* value of $\sum_{i=1}^{m} p'[i]y_i$ [15]. Note that finding such a pattern is equivalent to solving an unbounded knapsack problem where B is the knapsack size, the s_i's are the item sizes, and the y_i's are the item values. We thus have the following procedure for solving the original LP.

1. Use FFD to generate an initial set of patterns P.

2. Solve the LP based on pattern set P.
3. While not done do the following:
   (a) Solve the unbounded knapsack problem induced by the current LP.
   (b) If the resulting pattern has value 1 or less, we are done.
   (c) Otherwise add the pattern to P and solve the resulting LP.
4. Derive a packing from the current LP solution by rounding down.

The original approach of [14, 15] did not solve the LP in step (3a) but simply performed a single pivot. However, the reduction in iterations obtained by actually solving the LP's more than pays for itself, and this is the approach taken by current implementations.

There are still several potential computational bottlenecks here: (1) We have no subexponential bound on the number of iterations. (2) Even though modern simplex-based LP codes in practice seem to take time bounded by low order polynomials in the size of the LP, this is not a worst-case guarantee. (3) The unbounded knapsack problem is itself NP-hard.

Fortunately, there are ways to deal with this last problem. First, unbounded knapsack problems can be solved in (pseudo-polynomial) time $O(mB)$ using dynamic programming. This approach was proposed in the first Gilmore-Gomory paper [14]. A second approach was suggested in the follow-up paper [15], where Gilmore and Gomory observed that dynamic programming could often be bested by a straightforward branch-and-bound algorithm (even though the worst-case running time for the latter is exponential rather than pseudo-polynomial). The current common wisdom [2, 21] is that the branch-and-bound approach is to be preferred, and this is indeed the case for the small values of m considered in previous studies. In this paper we study the effect of using relatively straightforward implementations of both approaches, and conclude that solving the knapsack problems is not typically the computational bottleneck.

Note that the times for both the LP and the knapsack problems are almost independent of the number of items n. Moreover, the initial set of patterns can be found in time $O(m^2)$ by generating the FFD packing in size-by-size fashion rather than item-by-item. The same bound applies to our rounding procedure. Thus for any fixed discrete distribution, the Gilmore-Gomory approach should be asymptotically much faster than any of the classical online bin packing heuristics, all of which pack item-by-item and hence have running times that are $\Omega(n)$. Of course, in applications where the items must actually be individually assigned to bins, *any* algorithm must be $\Omega(n)$, but in many situations we are only looking for a packing plan or for bounds on the number of bins needed.

3 The Flow-Based Approach

An alternative approach to computing the LP bound, one that models the problem in a flow-oriented way, has recently been proposed independently by Valério de Carvalho [8, 9] and Csirik et al. [6, 7]. We follow the details of the latter formulation, although both embody the same basic idea. Let us view the packing process as placing items into bins one at a time. For each pair of an item size s and a bin level h, $0 \le h < B$, we have a variable $v(i,h)$, intended to represent the number of items of size s_i that are placed into bins whose prior contents totaled h. It is easy to see that the following linear program has the same optimal solution value as the one in the previous section: Minimize $\sum_{i=1}^{m} v(i,0)$, i.e., the total number of bins used, subject to

$$v(i,h) \ge 0, \qquad 1 \le i \le m, \quad 0 \le h < B \tag{3.1}$$

$$v(i,h) = 0, \qquad 1 \le i \le m, \quad s_i > B - h \tag{3.2}$$

$$v(i,h) = 0, \qquad 1 \le h < s_i, \quad 1 \le i \le m \tag{3.3}$$

$$\sum_{h=0}^{B-1} v(i,h) = n_i, \qquad 1 \le i \le m \tag{3.4}$$

$$\sum_{i=1}^{m} v(i,h) \le \sum_{i=1}^{m} v(i,h-s_i), \qquad 1 \le h < B \tag{3.5}$$

where the value of $v(k,h-s_k)$ when $h - s_k < 0$ is taken to be 0 by definition for all k. Constraints of type (3.2) say that no item can go into a bin that is too full to have room for it. Constraints of type (3.3) imply that the first item to go in any bin must be larger than the second. (This is not strictly necessary, but helps reduce the number of nonzeros in the coefficient matrix and thus speed up the code.) Constraints of type (3.4) say that all items must be packed. Constraints of type (3.5) say that bins with a given level are created at least as fast as they disappear.

Solving the above LP does not directly yield a packing, but one can derive a corresponding set of packing patterns using a simple greedy procedure that will be described in the full paper. Surprisingly, this procedure obeys the same bound on the number of non-zero patterns as does the classical column-generation approach, even though here the LP has $m + B$ constraints. In the full paper we prove the following:

THEOREM 3.1. *The greedy procedure for extracting patterns from a solution to the flow-based LP runs in time $O(mB)$ and finds a set C of patterns, $|C| \leq m$, that provides an optimal solution to the pattern-based LP.*

The flow-based approach has a theoretical advantage over the column-based approach in that it can be implemented to run in provably pseudo-polynomial time using the ellipsoid method. However, the LP involved is much bigger than the initial LP in the pattern-based approach, and it will have $\Theta(mB)$ nonzeros, whereas the latter will have $O(m^2)$ ($O(m)$ when the smallest item size exceeds cB for some fixed $c > 0$). Thus the pattern-based approach may well be faster in practice, even though $\Omega(m)$ distinct LP's may need to be solved.

4 Speedups by Grouping

A key idea introduced in the asymptotic approximation scheme for bin packing of [12] is that of *grouping*. Suppose the items in list L are ordered so that $s(a_1) \geq s(a_2) \geq \cdots \geq s(a_n)$, suppose $g << n$, and let $K = \lceil n/g \rceil$. Partition the items into groups G_k, $1 \leq k \leq K$, where $G_k = \{a_{g(k-1)+i} : 1 \leq i \leq g\}$ for $k < K$, and $G_K = \{a_i : g(K-1) < i \leq n\}$.

Consider the list L_1 consisting of g items of size $s(a_{gk+1})$, $1 \leq k < K$. There is a one-to-one correspondence between the items of L_1 and items at least as large in $\cup_{k=1}^{K-1} G_k$, so $OPT(L) \geq OPT(L_1)$. On the other hand, there is also a one-to-one correspondence between items in L with items in $L_2 = L_1 \cup G_1$ that are at least as large, so $OPT(L) \leq OPT(L_1) + g$. Thus if we use one of the previous two LP-based approaches to pack L_1, replace each item of size $s(a_{gk+1})$ in L_1 by an item from G_{k+1} in L, and then place the items of G_1 into separate bins, we can derive a packing of L that uses at most $OPT(L) + \lceil n/g \rceil + g$ bins. Varying g yields a tradeoff between running time and packing quality.

Moreover we can get better packings in practice as follows: After computing the fractional LP solution for L_1, round down the fractional patterns of L_1, obtaining a packing of subset of L_1, replace the items of size $s(a_{gk+1})$ in this packing by the *largest* items in G_{k+1}, and then pack the leftover items from L (including those from G_1) using FFD.

Note that by the construction of L_1, the solution of the LP for it will be a lower bound on $LP(L)$. Thus running this approach provides both a packing and a certificate for how good it is, just as does the original Gilmore-Gomory algorithm.

5 Instance Classes

In this study we considered three distinct classes of discrete distributions for generating our test instances: (1) *Discrete Uniform Distributions*, (2) *Near Uniform Sampled Distributions*, and (3) *Zipf's Law Sampled Distributions*.

5.1 Discrete Uniform Distributions. These are distributions denoted by $U\{h, j, k\}$, $1 \leq h \leq j < k$, in which the bin size $B = k$ and the item sizes are the integers s with $h \leq s \leq j$, all equally likely. Of particular interest is the special case where $h = 1$, which has been studied extensively from a theoretical point of view, e.g., see [3, 4, 6].

Let $L_n(F)$ denote a random n-item list with item sizes chosen independently according to distribution F, let $s(L)$ denote the lower bound on $OPT(L)$ obtained by dividing the sum of the item sizes in L by the bin size, and let $A(L)$ be the number of bins used in the packing of L generated by algorithm A. Define

$$EW_n^A(F) = E\big[A(L_n(F)) - s(L_n(F))\big]$$

Then we know from [4] that $EW_n^{OPT}(F)$ is $O(1)$ for all $U\{1, j, k\}$ with $j < k - 1$ and $\Theta(\sqrt{n})$ for $j = k - 1$. The same holds if OPT is replaced by the online Sum-of-Squares algorithm of [6, 7] (denoted by SS in what follows). On the other hand, although $EW_n^{FFD}(F) = O(1)$ for many of these distributions, it can also be $\Theta(n)$, albeit with a small constant of proportionality [3, 5].

Experimental studies of classical bin packing algorithms reported in [3, 7] concentrated on discrete uniform distributions with $k \leq 100$, the most thorough study being the one in [7] for $U\{1, j, 100\}$, $2 \leq j \leq 99$, and $U\{18, j, 100\}$, $19 \leq j \leq 99$. As we shall see, these present no serious challenge to our LP-based approaches. To examine questions of scaling, we also consider distributions that might arise if we were seeking better and better approximations to the continuous uniform distributions $U(0, \alpha]$, where item sizes are chosen uniformly from the real interval $(0, \alpha]$. For example the continuous distribution $U(0, .4]$ can be viewed as the limit of the sequence $U\{1, 200h, 500h\}$ as $h \to \infty$.

5.2 Bounded Probability Sampled Distributions. These distributions were introduced in [10], expanding on an instance generator introduced in [13], and are themselves randomly generated. To get a distribution of *type* $BS\{h, j, k, m\}$, $1 \leq h \leq j < k$ and $m \leq j - h + 1$ we randomly choose m distinct sizes s such that $h \leq s \leq j$, and to each we randomly assign a *weight* $w(s) \in [0.1, 0.9]$. The probability associated with size s is then $w(s)$ divided by the sum of all the weights. Values of the bin size $B = k$ studied in [10] range up to 10,000, and values of m up to 100. To get an n-item instance of type $BS\{h, j, k, m\}$, we randomly generate a distribution of this type and then choose n

item sizes according to that distribution. We consider three general classes of these distributions in our scaling experiments, roughly of the form $BS\{1, B/2, B, m\}$, $BS\{B/6, B/2, B, m\}$, and $BS\{B/4, B/2, B, m\}$.

The first sequence mirrors the discrete uniform distributions $U\{1, B/2, B\}$. The last two model the situation where there are no really small items, with the third generating instances like those in the famous 3-PARTITION problem. These last two are also interesting since they are unlike the standard test cases previously used for evaluating knapsack algorithms.

5.3 Zipf's Law Sampled Distributions. This is a new class, analogous to the previous one, but with weights distributed according to Zipf's Law. In a type $ZS\{h, j, k, m\}$ distribution, m sizes are chosen as before. They are then randomly permuted as $s_1, s_2, \ldots, s_m$, and we set $w(s_i) = 1/i$, $1 \leq i \leq m$. We tested sequences of ZS distributions mirroring the BS distributions above.

6 Results

Our experiments were performed on a Silicon Graphics Power Challenge with 196 Mhz MIPS R10000 processors and 1 Megabyte 2nd level caches. This machine has 7.6 Gigabytes of main memory, shared by 28 of the above processors. The parallelism of the machine was exploited only for performing many individual experiments at the same time. The programs were written in `C` and compiled with the system compiler using -O3 code optimization and 32-bit word options. (Turning off optimization entirely caused the dynamic programming knapsack solutions to take 4 times as long, but had less of an effect on the other codes.) LP's were solved by calling `CPLEX 6.5`'s primal simplex code. (`CPLEX 6.5`'s dual and interior point codes were noncompetitive for our LP's.) The "presolve" option was turned on, which for some instances improved speed dramatically and never hurt much. In addition, for the pattern-based codes we set the undocumented `CPLEX` parameter `FASTMIP` to 1, which turns off matrix refactoring in successive calls and yields a substantial speedup. Instances were passed to `CPLEX` in sparse matrix format (another key to obtaining fast performance).

We should note here that the CPLEX running times reported by the operating system for our larger instances could vary substantially (by as much as a factor of 2) depending on the overall load of the machine. The times reported here are the faster ones obtained when the machine was relatively lightly loaded. Even so, the CPLEX times and the overall running times that include them are the least accurately reproducible statistics we report. Nevertheless, they still suffice to indicate rough trends in algorithmic performance.

Our branch-and-bound code for the unbounded knapsack was similar to the algorithm MTU1 of [18] in that we sped up the search for the next item size to add by keeping an auxiliary array that held for each item the smallest item with lower density y_i/s_i. As a further speed-up trick, we also kept an array which for each item gave the next smallest item with lower density.

For simplicity in what follows, we shall let PDP and PBB denote the pattern-based approaches using dynamic programming and branch-and-bound respectively, and FLO denote the flow-based approach. A sampling of the data from our experiments is presented in Tables 1 through 10 and Figures 1 and 2. In the space remaining we will point out some of the more interesting highlights and summarize our main conclusions.

6.1 Discrete Uniform Distributions with $k = 100$. We ran the three LP-based codes plus FFD, BFD, and SS on five n-item lists generated according to the distributions $U\{1, j, 100\}$, $2 \leq j \leq 99$, and for $U\{18, j, 100\}$, $19 \leq j \leq 99$, for n = 100, 1,000, 10,000, 100,000, and 1,000,000. For these instances the rounded-down LP solutions were almost always optimal, using just $\lceil LP(L) \rceil$ bins, and never used more than $\lceil LP(L) \rceil + 1$. (No instance of bin packing has yet been discovered for which $OPT > \lceil LP(L) \rceil + 1$, which has led some to conjecture that this is the worst possible [16, 17, 20].) The value of rounding down rather than up is already clear here, as rounding up (when $n = 1,000,000$) yielded packings that averaged around $\lceil LP \rceil + 12$ for PBB and PDP, and $\lceil LP \rceil + 16$ for FLO. This difference can perhaps be explained by the observation that FLO on average had 45% more fractional patterns than the other two, something that makes more of a difference for rounding up than down.

Table 1 reports average running times for the first set of distributions as a function of n and j for PDP and FLO. (The times for PBB are here essentially the same as those for PDP.) The running times for all three codes were always under a second per instance, so in practice it wouldn't make much difference which code one chose even though FLO is substantially slower than the other two. However, one trend is worth remarking.

For the larger values of m, the average running times for PBB and PDP actually *decrease* as n increases, with more than a factor of 2 difference between the times for $n = 100$ and for $n = 10,000$ when $60 \leq m \leq 69$. This is a reproducible phenomenon and we will examine possible explanations in the next section.

As to traditional bin packing heuristics, for these distributions both FFD and SS have bounded expected excess except for $m = 99$. However, while FFD is almost as good as our LP approaches, finding optimal solutions

almost as frequently, SS is much worse. For instance, for $j = 90$, its asymptotic *average* excess appears to be something like 50 bins. Both these classical heuristics perform much more poorly on some of the $U\{18, j, 100\}$ distributions. Many of these distributions have $EW_n^{OPT}(F) = \Theta(n)$, and for these FFD and SS can use as many as 1.1 times the optimal number of bins (a linear rather than an additive excess).

6.2 How can an algorithm take less time when n increases? In the previous section we observed that for discrete uniform distributions $U\{1, j, 100\}$, the running times for PBB and PDP *decrease* as n increases from 100 to 10,000. This phenomenon is not restricted to small bin sizes, as is shown in Table 2, the top half of which covers the distribution $U\{1, 600, 1000\}$, with n increasing by factors of roughly $\sqrt{10}$ from m to $1000m$. Here the total running time consistently decreases as n goes up, except for a slight increase on the first size increment.

What is the explanation? It is clear from the data that the dominant factor in the running time decrease is the reduction in the number of iterations as n increases. But why does this happen? A first guess might be that this is due to numeric precision issues. `CPLEX` does its computations in floating point arithmetic with the default tolerance set at 10^{-6} and a maximum tolerance setting of 10^{-9}. Thus, in order to guarantee termination, our code has to halt as soon as the current knapsack solution has value $1 + \epsilon$, where ϵ is the chosen `CPLEX` tolerance. Given that the FFD packings for this distribution are typically within 1 bin of optimal, the initial error gets closer to the tolerance as n increases, and so the code might be more likely to halt prematurely as n increases.

This hypothesis unfortunately is unsupported by our data. For these instances, the smallest knapsack solutions that exceed 1 also exceed $1 + 10^{-4}$, and our pattern-based codes typically get the same solution value and same number of iterations whether ϵ is set to 10^{-4} or 10^{-9}. Moreover the solutions appear typically to be the true (infinite precision) optima. This was confirmed in limited tests with an infinite-precision Gilmore-Gomory implementation that combines our code with (1) the exact LP solver of Applegate and Still [1] (a research prototype that stores all numbers as rationals with arbitrary precision numerators and denominators) and (2) an exact dynamic programming knapsack code. Thus precision does not appear to be an issue, although for safety we set $\epsilon = 10^{-9}$ in all our subsequent experiments.

We suspect that the reduction in iterations as n increases is actually explained by the number of initial patterns provided by the FFD packing. As reported in Table 2, when $n = 600{,}000$ the FFD supplied patterns are almost precisely what is needed for the final LP – only a few iterations are needed to complete the set. However, for small n far fewer patterns are generated. This means that more iterations are needed in order to generate the full set of patterns needed for the final LP. This phenomenon is enough to counterbalance the fact that for the smallest n we get fewer item sizes and hence smaller LP's. The latter effect dominates behavior for distributions where FFD is not so effective, as shown in the lower part of Table 2, which covers the bounded probability sampled distribution $BS\{1, 6000, 10000, 400\}$. Here the number of excess FFD bins, although small, appears to grow linearly with n, and the total PDP running time is essentially independent of n, except for the smallest value, where less than 60% of the sizes are present.

6.3 How Performance Scales. In the previous section we considered how performance scales with n. Our next set of experiments addressed the question of how performance scales with m and B, together and separately. Since we are interested mainly in trends, we typically tested just one instance for each combination of distribution, m, and B, but this was enough to support several general conclusions.

Tables 3 and 4 address the case in which both m and B are growing. (B must grow if m is to grow arbitrarily.) Table 3 covers the discrete uniform distributions $U\{1, 200h, 500h\}$ for $h = 1, 2, 4, 8, 16, 32$. In light of the discussion in the last section, we chose a small fixed ratio of n to m ($n = 2m$) so as to filter out the effect of n and obtain instances yielding nontrivial numbers of iterations for PDP and PBB. Had we chosen $n = 1000m$, we could have solved much larger instances. For example, with this choice of n, PBB finds an optimal solution to an instance of $U\{1, 51200, 128000\}$ in just 36 iterations and 303 seconds.

For the instances covered by Table 3, the rounded down LP solution was always an optimal packing, as indeed was the FFD packing used to generate the initial set of patterns. In fact, the FFD solution always equaled the size bound $\lceil (\sum_{a \in L} s(a))/B \rceil$, so one could have concluded that the FFD packing was optimal without these computations. Nevertheless, it is interesting to observe how the running times for the LP-based codes scale, since, as remarked above, there are $U\{1, j, k\}$ distributions for which FFD's expected excess grows linearly with n, and for these the LP-based algorithms would find better packings. The times reported for PDP are roughly consistent with the combinatorial counts. The number of arithmetic operations needed for solving

the knapsack problems using our dynamic programming code grows as $\Theta(mB)$ (and so greater increases here suggest that memory hierarchy effects are beginning to have an impact). The time for solving an LP might be expected to grow roughly as the number of columns (patterns) times the number of pivots. Using "iterations" as a reasonable surrogate for the number of patterns, we get that overall time for PDP should grow as

$$(iterations)\big((iterations \times pivots) + mB\big)$$

Note that both iterations and pivots per LP are growing superlinearly, and so we would expect greater-than-cubic overall time, which is what we see (the times reported in successive rows go up by more than a factor of 8). The growth rate is still less than n^4, however. PBB is here faster than PDP since the knapsack time is comparatively negligible, although its advantage over PDP is limited by the fact that LP time has become the dominant factor by the time $B = 16,000$. It is also worth noting that for an individual instance the number of pivots per LP can be highly variable, as illustrated in Figure 1. The difficulties of the LP's can also vary significantly between PBB and PDP, whose paths may diverge because of ties for the best knapsack solution. For the instance depicted in Figure 1 the average number of pivots under PBB was 18% lower than that for PDP, although the same irregularity manifested itself. The extremely high numbers of pivots for some of the LP's in the PDP run suggest that the danger of runaway LP-time cannot be ignored, no matter what our average-case projections say. FLO's running times are again not competitive, and in any case its much larger memory requirement rules out applying it to the largest instances.

Table 4 contains analogous results for bounded probability distributions in which the sizes sampled must lie in the intervals $(0, B/2)$, $(B/6, B/2)$, or $(B/4, B/2)$. Once again, overall running times grow at a rate somewhere between n^3 and n^4 and LP time dominates dynamic programming time for the largest values of B. For the last set of distributions, however, LP time is exceeded by branch-and-bound knapsack solution time, which gets worse as the lower bound on the size interval increases. Indeed, for the $(B/4, B/2)$ set of distributions, the time per branch-and-bound knapsack solution closely tracks the time needed for full exhaustive search, i.e., $\Theta(m^3)$ in this case, and PBB is slower than FLO for m as large as $16,000$.

Another difference between the last two sets of distributions and the first lies in the "excess" of the rounded-down packing, i.e., the difference between the number of bins contained in that packing and the LP solution value. The first set of distributions behaves much like the discrete uniform distributions it resembles, with typical excesses of less than one. For the latter two, the excesses grow with m, although they are typically between 3 and 4% of m, far below the proven upper bound of m itself. It remains to be seen whether the true optimum number of bins is closer to the LP lower bound on the rounded-down upper bound.

Tables 5 and 6 cover experiments in which m was held fixed and B was allowed to grow. Here growth in dynamic programming time is expected, but note that branch-and-bound knapsack time also increases, perhaps because as B increases there are fewer ties and so more possibilities must be explored. Iterations also increase (perhaps because greater precision is now needed for an optimal solution), although pivots and seconds per LP remail relatively stable once a moderate value of B has been attained.

Table 7 shows the effect of increasing m while holding B fixed. Once again LP time eventually dominates dynamic programming time. In the $(B/2, B/4)$ case, FLO time again comes to dominate PBB time, and is even gaining on PDP as m approaches its maximum possible value, but it is not clear that we will ever find a situation where it beats the latter. PBB does have *one* surprising advantage over PDP in the $(B/2, B/4)$ case. As indicated in Table 8, the patterns generated by branch-and-bound knapsack solutions seem to be better in the context of the overall algorithm. PDP needs both more iterations and more pivots per iteration than does PBB. This doesn't hold for all distributions, but was seen often enough in our experiments to be suggestive.

Table 9 provides more detailed information for the $(B/6, B/2)$ case, illustrating the high variability in the branch-and-bound times, which not only can vary widely for the same value of m, but can actually decline as m increases. Figure 2 charts the evolution of LP time and branch-and-bound knapsack time during the run for one of the more troublesome instances. Note that here LP time is relatively well-behaved (in contrast to the situation charted in Figure 1), while branch-and-bound time now can vary widely depending on the stage of the overall computation.

6.4 Grouping. See Table 10. Here is a major surprise: For instances with $n \leq 10,000$ and $m = 1,600$, grouping not only yields running times that are orders of magnitude faster than those for the basic Gilmore-Gomory ($g = 1$) procedure, it also provides better packings. This is presumably because for this value of n and these values of g, the savings due to having far fewer patterns (and hence far fewer fractional patterns to round down) can outweigh the cost of having to separately pack the g largest items (which FFD does

fairly efficiently anyway). Even for $n = 1,000,000$, where $g = 1$ is now dominant in terms of solution quality, very good results can be obtained in very little time if $n/g \in \{100, 200\}$. Similar results hold for $m = 3,200$.

6.5 Zipf's Law Distributions. We do not have space here to present our results for ZS distributions, except to note that although they typically yielded similar behavior to that for the corresponding BS distributions, a few ZS instances caused more dramatic running time explosions than we have seen so far. In particular, for a million-city $ZS\{1667, 4999, 10000, 2200\}$ instance, the first 40 iterations of PBB (out of 7802) averaged over 24 minutes per knapsack solution and took roughly 60% of the total time.

6.6 Directions for Future Research. These preliminary results are based on straightforward implementations of the algorithmic components. Presumably we can improve performance by improving those components. One way to attack LP time, the major asymptotic bottleneck, would be to identify and remove unnecessary columns from the later LP's, rather than let the LP size grow linearly with iteration count. There are also more sophisticated knapsack algorithms to try, such as those of [18, 19]. Even a simple improvement to the dynamic programming code such as identifying and removing "dominated" items can have a major effect, and can be implemented by a relatively minor change in the inner loop of the code. Preliminary experiments suggest that this idea can often reduce dynamic programming time by a factor of 3 or more, as we shall illustrate in the full paper.

References

[1] D. L. Applegate and C. Still. Personal communication, 2002.

[2] V. Chvátal. The cutting-stock problem. In *Linear Programming*, pages 195–212. W. H. Freeman and Company, New York, 1983.

[3] E. G. Coffman, Jr., C. Courcoubetis, M. R. Garey, D. S. Johnson, L. A. McGeoch, P. W. Shor, R. R. Weber, and M. Yannakakis. Fundamental discrepancies between average-case analyses under discrete and continuous distributions. In *Proceedings 23rd Annual ACM Symposium on Theory of Computing*, pages 230–240, New York, 1991. ACM Press.

[4] E. G. Coffman, Jr., C. Courcoubetis, M. R. Garey, D. S. Johnson, P. W. Shor, R. R. Weber, and M. Yannakakis. Bin packing with discrete item sizes, Part I: Perfect packing theorems and the average case behavior of optimal packings. *SIAM J. Disc. Math.*, 13:384–402, 2000.

[5] E. G. Coffman, Jr., D. S. Johnson, L. A. McGeoch, P. W. Shor, and R. R. Weber. Bin packing with discrete item sizes, Part III: Average case behavior of FFD and BFD. (In preparation).

[6] J. Csirik, D. S. Johnson, C. Kenyon, J. B. Orlin, P. W. Shor, and R. R. Weber. On the sum-of-squares algorithm for bin packing. In *Proceedings of the 32nd Annual ACM Symposium on the Theory of Computing*, pages 208–217, New York, 2000. ACM.

[7] J. Csirik, D. S. Johnson, C. Kenyon, P. W. Shor, and R. R. Weber. A self organizing bin packing heuristic. In M. Goodrich and C. C. McGeoch, editors, *Proceedings 1999 Workshop on Algorithm Engineering and Experimentation*, pages 246–265, Berlin, 1999. Lecture Notes in Computer Science 1619, Springer-Verlag.

[8] J. M. Valério de Carvalho. Exact solutions of bin-packing problems using column generation and branch and bound. *Annals of Operations Research*, 86:629–659, 1999.

[9] J. M. Valério de Carvalho. Lp models for bin packing and cutting stock problems. *European Journal of Operational Research*, 141:2:253–273, 2002.

[10] Z. Degraeve and M. Peeters. Optimal integer solutions to industrial cutting stock problems: Part 2: Benchmark results. *INFORMS J. Comput.*, 2002. (To appear).

[11] Z. Degraeve and L. Shrage. Optimal integer solutions to industrial cutting stock problems. *INFORMS J. Comput.*, 11:4:406–419, 1999.

[12] W. Fernandez de la Vega and G. S. Lueker. Bin packing can be solved within $1+\epsilon$ in linear time. *Combinatorica*, 1:349–355, 1981.

[13] T. Gau and G. Wäscher. Cutgen1: A problem generator for the standard one-dimensional cutting stock problem. *European J. of Oper. Res.*, 84:572–579, 1995.

[14] P. C. Gilmore and R. E. Gomory. A linear programming approach to the cutting stock problem. *Oper. Res.*, 9:948–859, 1961.

[15] P. C. Gilmore and R. E. Gomory. A linear programming approach to the cutting stock program — Part II. *Oper. Res.*, 11:863–888, 1963.

[16] O. Marcotte. The cutting stock problem and integer rounding. *Math. Programming*, 33:82–92, 1985.

[17] O. Marcotte. An instance of the cutting stock problem for which the rounding property does not hold. *Oper. Res. Lett.*, 4:239–243, 1986.

[18] S. Martello and P. Toth. *Knapsack Problems.* John Wiley & Sons, Chichester, 1990.

[19] D. Pisinger. A minimal algorithm for the bounded knapsack problem. *INFORMS J. Computing*, 12:75–84, 2000.

[20] G. Scheithauer and J. Terno. Theoretical investigations on the modified integer round-up property for the one-dimensional cutting stock problem. *Oper. Res. Lett.*, 20:93–100, 1997.

[21] G. Wäscher and T. Gau. Heuristics for the integer one-dimensional cutting stock problem: A computational study. *OR Spektrum*, 18:131–144, 1996.

	PDP			FLO		
n	$30 \le j \le 39$	$60 \le j \le 69$	$90 \le j \le 99$	$30 \le j \le 39$	$60 \le j \le 69$	$90 \le j \le 99$
10^2	.040	.082	.058	.144	.182	.173
10^3	.033	.044	.050	.150	.206	.300
10^4	.030	.034	.041	.150	.206	.342
10^5	.029	.032	.035	.146	.203	.367
10^6	.029	.032	.034	.147	.204	.356

Table 1: Average running times in seconds for discrete uniform distributions $U\{1, j, 100\}$ as a function of j and n. Averages are taken over 5 samples for each value of j and n. Results for PBB are similar to those for PDP. Packings under all three approaches were almost always optimal.

$U\{1, 600, 1000\}$

	Ave#		#Pat		Pivots	Ave Secs		Tot	Opt	FFD
n	sizes	Iters	FFD	Final	/iter	LP	KNP	Secs	Val	Excess
600	374.7	730.7	170.3	901.0	18.7	.01	.01	17.4	180.9	.8
1,897	573.7	599.7	394.0	993.7	18.7	.02	.02	19.6	571.3	.7
6,000	600.0	157.0	633.7	790.7	34.1	.03	.02	7.2	1797.3	.7
18,974	600.0	77.3	800.0	877.3	51.0	.04	.02	4.3	5686.5	.5
60,000	600.0	46.7	881.3	928.0	58.7	.04	.02	2.6	18058.4	.6
189,737	600.0	25.0	885.7	910.7	18.8	.01	.02	.7	56941.8	.6
600,000	600.0	7.3	909.0	916.3	12.5	.00	.02	.2	180328.2	.5

$BS\{1, 6000, 10000, 400\}$

	Ave#		#Pat		Pivots	Ave Secs		Tot	Opt	FFD
n	sizes	Iters	FFD	Final	/iter	LP	KNP	Secs	Val	Excess
400	231.0	904.3	110.3	1014.7	27.9	.02	.07	80	117	1
1,264	355.3	1303.0	259.7	1562.7	45.0	.05	.11	201	404	1
4,000	394.7	1069.0	442.3	1511.3	55.4	.06	.12	202	1177	3
12,649	400.0	994.3	519.7	1514.0	57.8	.07	.12	194	3645	6
40,000	400.0	989.0	561.3	1550.3	58.0	.07	.13	199	11727	21
126,491	400.0	998.0	565.7	1563.7	58.0	.07	.12	193	38977	93
400,000	400.0	1014.7	576.7	1591.3	58.7	.07	.13	204	117241	197

Table 2: Effect of increasing N (by factors of roughly $\sqrt{10}$) on PDP, averaged over three samples for each value of N. For the BS table, three distinct distributions were chosen and we generated one sample for each distribution and each value of N.

$U\{1, 200h, 500h\}$, $h = 1, 2, 4, \ldots, 64$, $n = 2m$

			Pivots	Ave LP	Avg knp secs		Total secs		
m	B	Iters	/iter	secs	PDP	PBB	PDP	PBB	FLO
200	500	175	2.9	.00	.00	.00	.8	.3	13
400	1000	440	4.9	.00	.01	.00	6.7	2.0	156
800	2000	1011	10.9	.01	.04	.02	57.5	28.8	2167
1600	4000	2055	24.0	.07	.20	.00	565.6	194.4	38285
3200	8000	4667	57.0	.52	.91	.01	6669.1	2415.3	—
6400	16000	10192	202.8	4.21	3.78	.02	81497.7	41088.6	—

Table 3: Scaling behavior for LP-based codes. The number of distinct sizes in the instances was 87±1% of m and the number of initial patterns was 39±1% of m. Unless otherwise specified, entries are for PDP.

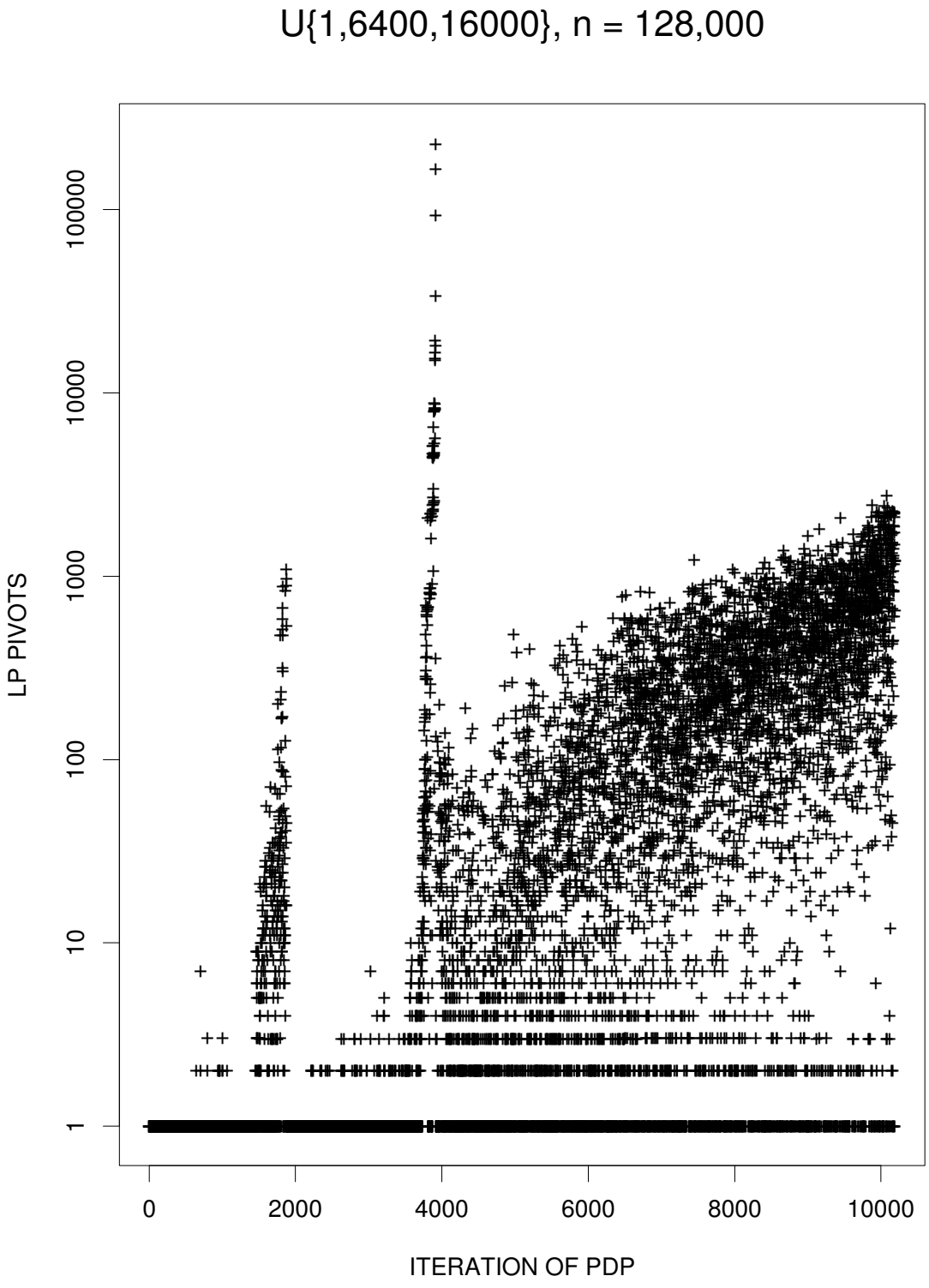

Figure 1: Number of Pivots for successive LP's under PDP plotted on a log scale. Over 15% of the total PDP running time was spent solving the 5 hardest LP's (out of 10,192). The possibility of such aberrant behavior means that the asymptotic running time projections derived from our data are unlikely to be worst-case guarantees.

$BS\{1, \lceil 625k/2 \rceil - 1, 625k, 100k\}$, $k = 1, 2, 4, 8, 16$, $n = 1,000,000$

			Pivots	Average	Avg knp secs		Total secs			PDP
m	B	Iters	/iter	LP secs	PDP	PBB	PDP	PBB	FLO	Excess
100	625	144	9.7	.00	.00	.00	1	1	12	.5
200	1250	238	17.9	.01	.01	.00	4	2	145	.6
400	2500	502	32.2	.03	.03	.00	30	16	2353	.8
800	5000	1044	69.3	.14	.12	.00	281	143	48620	.3
1600	10000	2154	166.0	1.11	.64	.01	3781	2898	—	.2
3200	20000	4617	385.4	10.39	2.72	.01	60530	38124	—	1.2

$BS\{\lfloor 625k/6 \rfloor + 1, \lceil 625k/2 \rceil - 1, 625k, 100k\}$, $k = 1, 2, 4, 8, 16$, $n = 1,000,000$

			Pivots	Average	Avg knp secs		Total secs			PDP
m	B	Iters	/iter	LP secs	PDP	PBB	PDP	PBB	FLO	Excess
100	625	184	10.4	.00	.00	.00	1	1	4	2.6
200	1250	375	21.2	.01	.01	.00	7	4	41	4.8
400	2500	840	46.9	.05	.03	.04	60	63	404	9.6
800	5000	1705	95.8	.23	.12	.51	597	1092	4523	18.0
1600	10000	3730	214.3	1.48	.53	.46	7527	5847	—	37.7
3200	20000	7845	478.5	10.76	2.34	5.02	102730	92778	—	76.2

$BS\{\lfloor 625k/4 \rfloor + 1, \lceil 625k/2 \rceil - 1, 625k, 100k\}$, $k = 1, 2, 4, 8, 16$, $n = 1,000,000$

			Pivots	Average	Avg knp secs		Total secs			PDP
m	B	Iters	/iter	LP secs	PDP	PBB	PDP	PBB	FLO	Excess
100	625	116	5.3	.00	.00	.00	0	1	2	3.0
200	1250	427	17.8	.01	.01	.03	7	14	12	11.2
400	2500	704	29.3	.02	.02	.17	33	107	101	16.5
800	5000	1422	52.5	.08	.11	1.08	274	1299	800	30.0
1600	10000	3055	119.8	.61	.47	8.07	3314	20123	19415	57.2
3200	20000	6957	265.9	3.59	2.16	67.73	40001	345830	—	128.5

Table 4: Scaling behavior for bounded probability sampled distributions. Unless otherwise specified, entries are for PDP. Note that amount by which the PDP packing exceeds the LP bound does not grow significantly with m in the first case, and is no more than 3% or 4% of m in the latter two.

$BS\{1,\ 625h-1,\ 1250h,\ 200\}$, $n = 1,000,000$

			Pivots	Ave LP	Ave knp secs		Total secs			PDP
h	B	Iters	/iter	secs	PDP	PBB	PDP	PBB	FLO	Excess
1	1,250	220	14.4	.01	.01	.00	3	3	12	.4
2	2,500	320	21.5	.01	.02	.00	9	5	145	.5
4	5,000	510	25.4	.02	.03	.00	24	9	2353	.8
8	10,000	444	26.0	.02	.06	.00	36	10	—	.3
16	20,000	600	29.0	.02	.13	.00	93	14	—	.3
32	40,000	736	28.0	.03	.28	.01	229	23	—	.4
64	80,000	776	29.9	.05	.58	.01	485	27	—	1.0
128	160,000	976	29.4	.04	1.16	.03	1170	57	—	.6
256	320,000	977	27.5	.03	2.88	.17	2834	205	—	.8
512	640,000	1081	28.7	.03	11.06	.19	11970	231	—	.5
1024	1,280,000	1267	32.3	.04	41.49	.67	52532	894	—	.8

Table 5: Effect of increasing the bin size B while m an N are fixed and other parameters remain proportionately the same (one instance for each value of h). Unless otherwise specified, column entries refer to PDP. The "Excess" values for all three algorithms are roughly the same.

$BS\{1250h+1,\ 2500h-1,\ 5000h,\ 1000\}$, $n = 1,000,000$

			Pivots	Ave LP	Ave knp secs		Total secs			PDP
h	B	Iters	/iter	secs	PDP	PBB	PDP	PBB	FLO	Excess
1	5,000	1778	70.7	.14	.12	1.43	470	2100	1212	36.9
2	10,000	2038	69.4	.15	.34	1.80	1002	3346	3050	37.8
4	20,000	2299	75.5	.19	.65	1.95	1925	4108	8957	37.4
8	40,000	2617	74.6	.19	1.35	2.13	4044	5243	29187	38.5
16	80,000	2985	80.6	.23	2.79	2.28	9019	6383	—	38.9
32	160,000	3195	71.5	.21	5.65	2.44	18732	7723	—	32.8

Table 6: Effect of increasing the bin size B while m and N are fixed and other parameters remain proportionately the same (one instance for each value of h). Unless otherwise specified, the entries are for PDP, which has roughly the same number of pivots as PBB but averages about 18% more iterations than PBB (the ratio declining as B increases). The "Excess" values for all three algorithms are roughly the same.

$BS\{1, 4999, 10000, m\}$, $n = 1,000,000$

m	Iters	Pivots /iter	Average LP secs	Avg knp secs PDP	Avg knp secs PBB	Total secs PDP	Total secs PBB	Total secs FLO	PDP Excess
100	411	12.0	.01	.03	.01	15	5	27131	1.2
200	453	29.3	.02	.06	.00	37	9	47749	.2
400	843	56.2	.07	.13	.00	169	65	—	.8
800	1454	98.3	.30	.29	.00	859	368	—	.6
1600	2326	157.4	1.65	.63	.01	5308	2443	—	1.2
3200	2166	212.6	4.66	1.28	.01	12872	5684	—	.6

$BS\{5001, 9999, 20000, m\}$, $n = 1,000,000$

m	Iters	Pivots /iter	Average LP secs	Avg knp secs PDP	Avg knp secs PBB	Total secs PDP	Total secs PBB	Total secs FLO	PDP Excess
100	203	6.3	.00	.05	.00	11	1	562	3.7
200	481	13.3	.01	.10	.02	53	12	1197	6.3
400	1121	27.6	.03	.22	.19	281	219	2879	13.8
800	1864	57.2	.11	.50	1.18	1131	2017	6745	31.5
1600	3586	116.2	.56	1.08	9.08	5878	26819	19415	63.9
3200	6957	265.9	3.59	2.16	67.73	40001	345830	—	128.5

Table 7: Effect of increasing the number of item sizes m while keeping the bin size fixed. Unless otherwise specified, the entries are for PDP.

	$BS\{1, 4999, 10000, m\}$			$BS\{5001, 9999, 20000, m\}$		
m	Iters	Pivots /iter	LP secs	Iters	Pivots /iter	LP secs
100	.95	.99	1.00	1.16	1.15	1.00
200	1.05	1.01	1.00	1.15	1.14	1.00
400	.98	1.00	1.00	1.11	1.05	1.50
800	1.04	1.01	1.15	1.18	1.07	1.10
1600	1.01	.99	.95	1.28	.98	1.10
3200	.97	1.05	1.00	1.42	.96	1.18
Average	1.00	1.01	1.02	1.22	1.06	1.15

Table 8: Ratios of statistics for PDP to those for PBB. Note that for the $BS\{5001, 9999, 20000, m\}$ distributions, the dynamic programming knapsack solver seems to be generating worse patterns than the branch-and-bound knapsack solver, leading to consistently more iterations and usually more pivots per iteration. This appears to be typical for $BS\{h, j, B, m\}$ distributions when h is sufficiently greater than 0.

$BS\{1667, 4999, 10000, m\}$

m	Iters PBB	Iters $\frac{\text{PDP}}{\text{PBB}}$	Ave Pivots PBB	Ave Pivots $\frac{\text{PDP}}{\text{PBB}}$	Ave LP secs PBB	Ave LP secs $\frac{\text{PDP}}{\text{PBB}}$	Avg knp secs PDP	Avg knp secs PBB	Total secs PDP	Total secs PBB	Total secs FLO	PDP Excess
100	262	1.03	10.9	.98	.00	1.00	.03	.00	8	2	644	1.8
200	659	1.06	20.4	1.15	.01	1.00	.05	.01	48	15	10833	4.6
400	1147	1.04	49.9	1.00	.06	1.00	.11	.01	205	82	58127	9.8
600	1477	1.11	71.9	1.00	.13	1.00	.17	.02	500	223	—	14.5
800	1864	1.12	101.4	.99	.26	1.19	.24	.03	1138	537	—	20.0
1000	2352	1.08	124.3	1.01	.44	1.16	.31	.75	2109	2805	—	24.2
1200	2618	1.11	136.5	1.02	.59	1.25	.38	2.67	3240	8534	—	29.0
1400	2902	1.14	184.8	1.01	1.01	1.27	.46	.21	5719	3567	—	33.4
1600	3146	1.19	212.1	1.01	1.40	1.06	.53	.46	7527	5847	—	37.7
1800	3433	1.19	234.8	.98	1.83	1.10	.62	.46	10810	7886	—	43.3
2000	3903	1.17	260.7	1.08	2.44	1.18	.70	1.98	16437	17259	—	47.3
2200	4221	1.15	291.1	1.02	3.16	1.08	.75	2.76	20240	24973	—	51.5
2400	4242	1.23	331.4	1.03	3.95	1.08	.81	1.01	26482	21025	—	54.7

Table 9: Results for bounded probability distributions and $n = 1,000,000$ (one sample for each value of n). Excesses for PBB and the flow-based code (where available) were much the same.

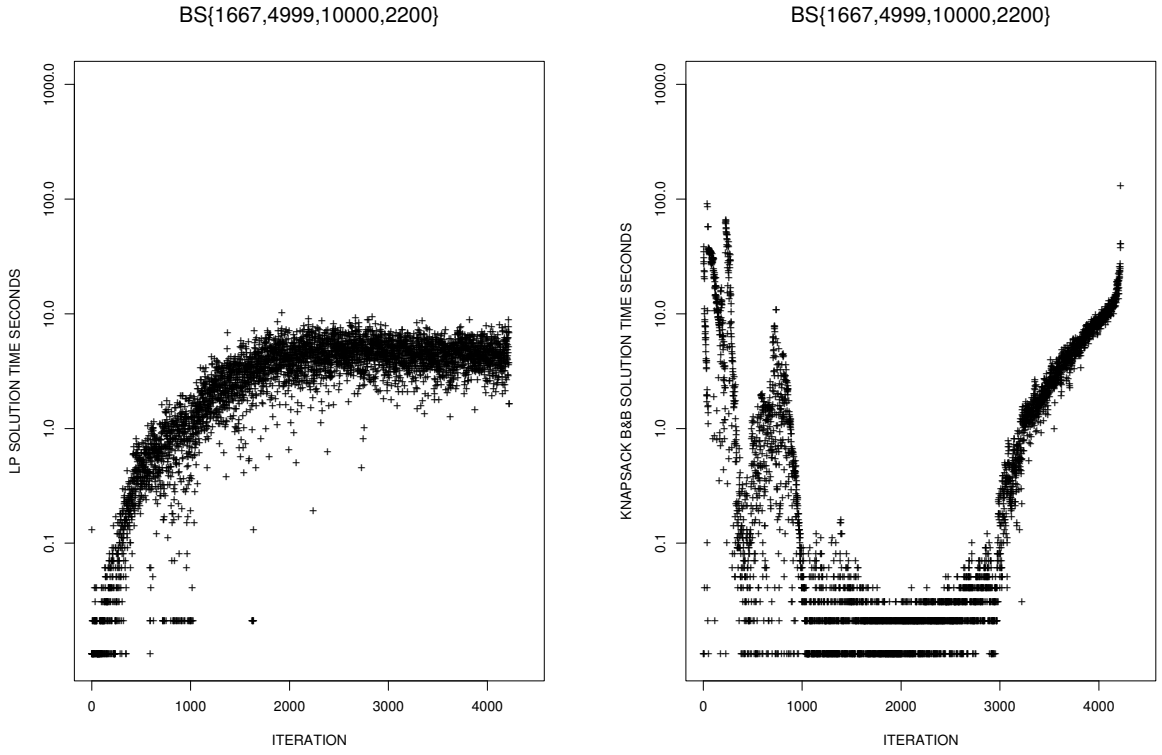

Figure 2: LP and Branch-and-Bound running times (plotted on a log scale) for successive iterations when PBB is applied to a million-item instance of type $BS\{1667, 4999, 10000, m\}$. Note that LP times here are much more well-behaved than they were for the instance of $U\{1, 6400, 16000\}$ covered in Figure 1. Now the potential for running time blow-up comes from the knapsack code, whose running time for $U\{1, 6400, 16000\}$ was negligible. The situation can be even worse for Zipf Law distributions.

$BS\{1667, 4999, 10000, 1600\}$					$BS\{5001, 9999, 20000, 1600\}$				
g	#Sizes	Total PDP Secs	Percent Packing Excess	Percent LB Shortfall	g	#Sizes	Total PDP Secs	Percent Packing Excess	Percent LB Shortfall
$n = 1,000$									
1	1600	853	4.580	.000	1	1600	1018	5.963	.000
5	200	39	1.675	.531	50	200	55	2.318	.471
10	100	7	1.094	.978	100	100	13	1.277	.923
20	50	2	1.385	1.879	200	50	2	1.277	1.744
	FFD	.02	5.161	.150		FFD	.03	7.265	3.147
$n = 10,000$									
1	1600	7185	.964	.000	1	1600	5681	1.471	.000
25	400	179	.392	.256	25	400	269	.440	.222
50	200	46	.362	.508	50	200	57	.518	.433
100	100	10	.542	1.004	100	100	10	.569	.813
200	50	2	1.084	1.949	200	50	2	1.085	1.556
	FFD	.2	4.908	.000		FFD	.2	7.451	3.506
$n = 100,000$									
1	1600	7627	.119	.000	1	1600	6117	.162	.000
250	400	195	.137	.253	250	400	265	.136	.218
500	200	46	.249	.505	500	200	57	.229	.432
1000	100	9	.516	.982	1000	100	11	.460	.830
2000	50	1	1.108	1.888	2000	50	2	.985	1.589
	FFD	.2	5.044	.000		FFD	.2	7.384	3.565
$n = 1,000,000$									
1	1600	7527	.011	.000	1	1600	5878	.017	.000
2500	400	197	.126	.251	2500	400	266	.110	.217
5000	200	48	.250	.502	5000	200	63	.216	.435
10000	100	9	.529	.976	10000	100	13	.444	.856
20000	50	1	1.123	1.780	20000	50	2	.954	1.644
	FFD	.2	5.008	.000		FFD	.2	7.310	3.142
	BF	134	6.320	–		BF	271	8.909	–
	SS	305	.478	–		SS	518	4.341	–

Table 10: Results for grouping. The "Percent Packing Excess" is the percent by which the number of bins used in the rounded down packing exceeds the LP lower bound. the "Percent LB Shortfall" is the percent by which the fractional solution for the grouped instance falls short of $LP(L)$. For comparisons purposes, we include results for an $O(m^2)$ implementation of FFD and, in the case of $n = 1,000,000$, for $O(nB)$ implementations of the online algorithms Best Fit (BF) and Sum-of-Squares (SS). The "Shortfall" entry for FFD gives the percent gap between $LP(L)$ and the size-based lower bound $s(L) = \sum_s(a_i)/B$.