Heuristics for Vector Bin Packing

Article

4 authors, including:

Rina Panigrahy
Google Inc.
127 PUBLICATIONS **7,659** CITATIONS

SEE PROFILE

Lincoln Uyeda
Microsoft
2 PUBLICATIONS **271** CITATIONS

SEE PROFILE

Udi Wieder
Microsoft
38 PUBLICATIONS **2,358** CITATIONS

SEE PROFILE

Heuristics for Vector Bin Packing

Rina Panigrahy[1], Kunal Talwar[1], Lincoln Uyeda[2], and Udi Wieder[1]

[1] Microsoft Research Silicon Valley. {rina,kunal,uwieder}@microsoft.com
[2] Microsoft's VMM product group. lincolnu@microsoft.com

Abstract. Inspired by virtual machine placement problems, we study heuristics for the Vector Bin Packing problem, where we are required to pack n items represented by d-dimensional vectors, into as few bins of size 1^d each as possible. We systematically study variants of the First Fit Decreasing (FFD) algorithm that have been proposed for this problem. Inspired by bad instances for FFD-type algorithms, we propose new geometric heuristics that run nearly as fast as FFD for reasonable values of n and d.

We report on empirical evaluations of the FFD-based, as well as the new heuristics on large families of distributions. We identify which FFD variants work best in most cases and show that our new heuristics usually outperform FFD-based heuristics and can sometimes reduce the number of bins used by up to ten percent. Further, in all cases where we were able to compute the optimal solution we found our new heuristics within few percent of optimal. We conclude that these new heuristics are an excellent alternative to FFD-based heuristics and are prime candidates to be used in practice.

1 Introduction

In the $d-$dimensional Vector Bin Packing Problem (VBP_d) we are given a set $\mathcal{I}$ of n items $I^1, I^2, ..., I^n$ where each $I^i \in \mathbb{R}^d$. A *valid packing* is a partition of $\mathcal{I}$ into k sets (or bins) $B_1, ..., B_k$ where for each bin j and for each dimension i, $\sum_{\ell \in B_j} I_i^\ell \leq 1$. The goal in the VBP_d problem is to find a valid packing while minimizing k. VBP_d is NP-hard for every d and APX-hard for $d \geq 2$.

Vector bin packing models well static resource allocation problems where there is a set of servers with known capacities and a set of services with known demands. The demands of services and the capacity of servers span across several dimensions. Recently there is renewed interest in the VBP problem because it models particulary well the problem of virtual machine placement. Virtualization has been a growing trend in the data-center with the promise of utilizing compute power more efficiently. Many businesses have started adopting this technology to save on IT budgets and maintenance costs. However the power of this technique is only as good as the management layer that schedules/assigns the virtual machines into a pool of machines in the data-center. When doing this assignment, it is important to ensure that no host gets overloaded while minimizing the number of hosts used. The goal is not only to reduce the upfront

investment in hardware, but to minimize the energy cost of operating the data center, even when hardware may have been over-provisioned.

This problem is made difficult by the multidimensional nature of load. For example, each virtual machine has its own CPU utilization, memory, and disk and network input and output requirements. Analogously, each host has a capacity across each of these dimensions, and the assignment should ensure that the number of hosts is minimized while no capacity constraint is violated. Moreover, often these requirements vary over time, and if we wish to avoid migration, we can model the problem by having a dimension for each resource, for each time epoch. This has the effect of further increasing the dimensionality of the problem.

If we assume that when different virtual machines are placed in the same host, their load across each dimension is accrued additively then the problem of statically assigning virtual machines to hosts is exactly the vector bin packing problem. It is therefore not surprising that algorithms for VBP had been proposed as early as four decades ago [15], [17] and heuristics have been implemented in systems that perform Virtual Machine consolidation.

Roughly speaking there are two types of algorithms that have been suggested for this problem. Perhaps the most natural set of heuristics for VBP are First Fit Decreasing (FFD) algorithms. In these heuristic there is some weight function applied to the items so that each item is assigned a single scalar. The items are then sorted and placed sequentially from large to small. For the case $d = 1$ this approach is known to be very effective, both formally and in practice. We discuss this family of heuristics in greater length in Section 2. Several systems and research prototypes have suggested and implemented variants of FFD (see related work section). Another approach is phrasing the problem as an Integer Linear Program (ILP) and then either solving the ILP with branch and bound methods (via some ILP solver) or solve the relaxed linear program, and employ some method for rounding the solution. This approach could yield good theoretical guarantees (see Section 1.1), however it has been our experience and the experience of others (c.f. [22]) that these algorithms do not scale beyond rather smallish instances. For n beyond a few hundred the LP-based algorithms in our experiments were prohibitively slow. Due to lack of space we omit a report on these experiments from this manuscript.

Our Contributions Our goal is to design and evaluate the most effective heuristics for VBP_d. We investigate various variants of FFD, and propose new geometric heuristics. We empirically evaluate these heuristics on synthetic data drawn from several distributions previously suggested in literature. We show that our new heuristics always do at least as well as the best FFD variants, and often noticeably improve on them. In cases that we were able to compute the truly optimal solution our heuristics proved to be within a few percent from optimal.

1.1 Related Work

The Vector bin packing problem is NP hard, even when restricted to the one-dimensional case, for which there is an *asymptotic PTAS*(see e.g. [24]). The

running time of this algorithm however is prohibitively large in practice. The most popular heuristics are FFD and its variants. There are many excellent books such as [13],[14] that detail most of the theory literature on bin packing problems.

For $d \geq 2$ the vector bin packing problem is known to be APX-hard[25] which means that there is no asymptotic PTAS for the problem, unless $P = NP$. Moreover, [5] show a $d^{\frac{1}{2}-\epsilon}$ hardness of approximation. For any $\epsilon > 0$, there is a classic $(d + \epsilon)$-approximation [11]. Fernandez de la Vega and Lueker [8] showed that one can get a $(d+\epsilon)$-approximation in time linear in n. Chekuri and Khanna showed a $(1 + d\epsilon + \ln(1/\epsilon))$-approximation which gives a $O(\ln d)$ approximation when d is constant. Bansal, Caprara and Sviridenko [2] improve this to get an $(1 + \ln d + \epsilon)$-approximation,for any $\epsilon > 0$. Both these algorithms run in time that is exponential in d (or worse). Yao [27] showed that no algorithm running in time $o(n \log n)$ can give better than a d-approximation.

In light of previous results it makes sense to study heuristics that may have worse formal guarantees but run well in practice. Maruyama, Chang and Tang [17] studied generalizations of one dimensional heuristics to the multidimensional case, and compared various variants. Spieksma [20] studied the two-dimensional version of the problem and gave lower bounding and heuristics for the problem. Han, Diehr and Cook [12] present heuristics and exact algorithm for a variant where the bins are not identical. Caprara and Toth [3] studied the 2-dimensional case in detail, and gave improved lower bounding techniques, heuristics, and exact algorithms for the problems. Many of these are tailored to the two-dimensional case. Finally, Stillwell et. al [22] studied variants of FFD concluding that the algorithm we call FFDAvgSum is best in practice. They also show that genetic algorithms don't perform well.

To the best of our knowledge, for the large dimensional case, no practical algorithms better than variants of FFD have been explored.

In general systems that manage resources in a shared hosting environment can all benefit from good heuristics for VBP (c.f. [4, 7, 6, 18]), there are far too many of those to be covered extensively here. Focusing on virtual machine placement there are several VM consolidation heuristics currently used in research prototypes and real VM management tools. We briefly outline how they are related to the heuristics described here. Sandpiper [26], a research system that enables live migration of VMs around overloaded hosts, uses a heuristic corresponding to *FFDProd*, taking the product of CPU, memory, and network loads. Another work from IBM research [23], an application placement controller for data centers, performs application assignment/consolidation handling resource demands for CPU and memory. This work combines the two dimensions into a scalar by taking the ratio of the CPU demand to the memory demand. Even though this heuristic does not fit into the *FFDAvgSum* or *FFDProd*, it shares the similar characteristic of ignoring complementary distributions of resource demands in different dimensions. Microsoft's Virtual Machine Manager [29] uses the Dot-Product and Norm-based Greedy, heuristics described below. The work [21] proposes to use Euclidean distance between resource de-

mands and residual capacity as a metric for consolidation, a heuristic analogous to Norm-based Greedy.

2 The FFD Family of Algorithms

One of the most natural heuristics for one dimensional bin packing is a greedy algorithm in which items are sorted by size in decreasing order and then items placed sequentially in the first bin that has sufficient capacity. This algorithm is often referred to as *First Fit Decreasing* (FFD). FFD is guaranteed to find an allocation with at most $\frac{11}{9}OPT + 1$ bins in the one dimensional case (c.f. [24]) and is known to be effective in practice.

There isn't a unique obvious way of generalizing FFD for the multi-dimensional case. One has to decide how to assign a *weight* to a d-dimensional vector. In this work we've tested two natural options:

$$w(I) = \prod_{i \leq d} I_i \qquad \text{(FFDProd)} \tag{1}$$

$$w(I) = \sum_{i \leq d} a_i I_i \qquad \text{(FFDSum)} \tag{2}$$

where the vector $\boldsymbol{a} = a_1, ...a_d$ is a scaling vector of our choosing. The vector $\boldsymbol{a}$ has two functions. First, it is needed to scale and normalize demand across dimensions (which not needed when taking the product of the demands). Secondly, it allows us to weight the demands according their importance, or their likelihood of actually being a bottleneck for placement. We test several ways of doing that.

Selecting the weights If the placement problem is effectively constrained by a single resource, say all VM's are CPU bound, then the problem is in essence one dimensional and FFD should probably be the algorithm of choice. Thus for example, if the demands in the first dimension always dominate the demands in the other dimension, the FFD variant should ignore the demands in the other dimensions. The FFD algorithm that uses the product weight (*FFDProd*) does not have this property: variations in all dimensions affect the ordering, irrespective of the importance of the dimension. *FFDSum* is more robust in this regard, the dimensions that are not scarce can be assigned smaller coefficients a_i's and have a smaller impact on the ordering. A natural choice is to take a_i to be equal to the average demand $avdem_i = \frac{1}{n}\sum_{\ell=1}^{n} I_i^{\ell}$ in dimension i. This leads to the heuristic that we call *FFDAvgSum*. Another option we explore is to take a_i to be exponential in $avdem_i$, i.e. $a_i = \exp(\epsilon \cdot avdem_i)$ for some suitable constant ϵ. We call this heuristic *FFDExpSum*. These weights are also used in the geometric heuristics described in Section 3. We observe that in [22] it is reported that FFDSum showed the best performance, however in their case the demands across the dimensions were sampled i.i.d and thus $\boldsymbol{a}$ was taken to be the all ones vector.

Running time of FFD Recall that FFD has two phases, in the first phase the items are sorted, which takes $O(nd+n\log n)$ time. In the second phase the items are placed sequentially in the *first bin* with sufficient remaining capacity. In the one dimensional case the total running time of the algorithm is bounded by $O(n\log n)$. Unfortunately, in the multi-dimensional case this is not known to be the case as the shape of the item determines whether a bin has sufficient capacity. An algorithm that scans all bins whenever an item is to be inserted has a running time of $\Omega(n\log n + nk)$, where k is the number of bins in the solution. This is the algorithm which we used in our implementation. An asymptotically faster algorithm can be derived by observing that each iteration we need to perform a d-dimensional orthogonal range-max operation. Using data structures designed for this problem (see e.g. [1]), one can get an $O(n\log^{d-2} n)$ time implementation. It is not clear whether these algorithms are more efficient than the naive n^2 algorithms for reasonable sized high dimensional data. For example, when $d = 6$ and $n = 1000$, $\log^{d-1} n$ is rougly 10^4, so that a quadratic time algorithm would already seem more efficient than the $O(n\log^{d-2} n)$ one.

Bad instances for FFD

It is easy to see that any greedy algorithm has an approximation ratio of at most $2d$. Since this is a weak guarantee it is important to identify inputs for which FFD performs particularly poor. In this section we identify such a family and argue that it is natural enough to motivate us to look for other algorithms.

Consider the two-dimensional instance where half the items are of size $(\frac{1}{3}, \frac{1}{6})$, and the other half of size $(\frac{1}{6}, \frac{1}{3})$. In this case, the optimal solution puts four items per bin, two of each kind, while any FFD variant would place three items per bin. This example can be easily generalized to give a worse class of instances.

Theorem 1. *For any integer k and large enough n there is an instance for which FFD partitions the n items into $\frac{n}{k}$ bins, while it is possible to partition the items into $\frac{n}{(k-1)d}$. Thus the approximation ratio of FFD is at least $(1-\frac{1}{k})d$.*

Proof. Assume we have items of d types. Every item I of type T_i has

$$I_j = \begin{cases} \frac{1}{k} & j = i \\ \frac{1}{(d-1)(k-1)k} & j \neq i \end{cases}$$

Assume we have exactly n/d items of each type. Observe that if we pack in each bin exactly $k-1$ items of each type then the load in each dimension is $\frac{k-1}{k} + \frac{(d-1)(k-1)}{(d-1)(k-1)k} = 1$, so OPT$\leq n/d(k-1)$. On the other hand, every FFD algorithm, irrespective of how exactly the weights are calculated would assign the same weight to all items of the same type, and thus would place all items of the same type one after the other (if items of two types have the same weight we can perturb them by a small amount to break the tie). Now, once k items of type T_i are placed in the same bin, dimension i is at full capacity and no more items can be placed. We conclude that FFD would pack the items into at least n/k items.

The distribution described above is somewhat contrived, but it is also quite robust. Observe that even if the demand vector of each item is perturbed by $O(1/dk^2)$, the approximation ratio of FFD is still $\Omega(d)$. Further, FFD would perform poorly even if we don't have *exactly* n/d items of each type. Given these observations we expect FFD not to perform particularly well when the input exhibits strong negative dependence across dimensions. This intuition is confirmed in our experiments.

3 Geometric heuristics

As the example in Theorem 1 demonstrates, FFD-based heuristics can be far from optimal. Instances similar to those are likely to arise in virtual machine placement settings: scientific computations may have high CPU requirements but low I/O requirements while web servers would behave in the opposite way. Similarly if the dimensions represent time epochs, then a service that has high demand during the day may have low demand during the night and vice versa. FFD based heuristics may fail to take advantage of such complementarities. To alleviate this, we propose a different generalization of FFD to multiple dimensions. The algorithms in the previous section generalize the *item-centric* view of FFD:

意思就是所有的item

不同维度变成一维的方式都是同等对待不好

Any definition of "size" in the multi-dimensional case will lead to a variant of FFD as in the last section. A different view of the (one-dimensional) FFD is what we call the *bin-centric* view:

FFD Item-centric
1. Sort items in decreasing order of "size".
2. **For** $\ell = 1$ to n **do**
2.1 Place item ℓ in first bin where it will fit.
2.2 **endfor**

Viewed this way, FFD has one bin open at any time, and in each time step, places the largest item that will fit the current bit. If there is no such item, the bin is declared closed, and we open a fresh bin. In one dimension these are implementations of exactly the same algorithm.

FFD Bin-centric
1. **While** there are items remaining to be placed **do**
1.1 Start a new bin
1.2 **While** some item fits in this bin **do**
1.2.1 Place "largest" remaining item that fits in the bin
1.2.2. **endwhile**
1.3.**end while**

In multiple dimensions, any definition of "largest" will lead to a generalization of FFD. We would like heuristics that takes into account not only the item's demand, but also how well it aligns with the remaining capacities in the open bin. We next suggest such generalizations.

Dot-Product (*DotProduct*): This heuristic defines "largest" to be the item that maximizes the dot product between the vector of remaining capacities and the vector of demands for the item. Formally, at time t let $r(t)$ denote the vector of *remaining* or *residual* capacities of the current open bin, i.e. subtract from

remaining应该指的是当前这个item放进去之前

the bin's capacity the total demand of all the items currently assigned to it. It places the item I^ℓ that maximizes the a-weighted dot product $\sum_i a_i I_i^\ell r(t)_i$ with the vector of remaining capacities without violating the capacity constraint. The weights a_i are calculated in the same manner as described in Section 2.

Norm-based Greedy (*L2*): This heuristic looks at the difference between the vectors I^ℓ and the residual capacity $r(t)$ under a certain norm, instead of the dot product. For example, for the ℓ_2 norm, from all unassigned items, it places the item I^ℓ that minimizes the quantity $\sum_i a_i (I_i^\ell - r(t)_i)^2$ and the assignment does not violate the capacity constraints. The weights a_i are again chosen as in Section 2. Similarly, using the $|\cdot|_1$ and $|\cdot|_\infty$ norms instead of the $|\cdot|_2$ norm gives us algorithms *L1* and *ELInf*.

Grasp and Bubblesearch The above heuristics can be augmented with *Grasp[k]* [10, 19] or *Bubblesearch* [16]. In *Grasp[k]*, instead of picking the best option at any time step, a random one from the best k is chosen. In Bubblesearch, the kth best is chosen with probability proportional to $(1-p)^k$, for an appropriate parameter p. Thus Step 1.2.1 is replaced by "Choose the kth "largest" item, where k is chosen from a geometric distribution with parameter p. One can run this random experiment `nruns` many times, and pick the allocation that uses the fewest number of bins.

Bad Example for our new heuristics We remark that the example of Theorem 1 can be modified to create an instance where the new proposed heuristics are off by the same factor. Indeed suppose that the items of type T_i were to be each split uniformly into $(dk)^i$ items of equal size, so that there are $\frac{n(dk)^i}{d}$ items of size $\frac{1}{(dk)^i}$ times the size of the type T_i items in Theorem 1. This instance forces our algorithms to make the same choices as FFD and results in the same approximation ratio. Note however that these new instances are significantly less robust to perturbations.

4 Experiments

We empirically evaluate our algorithms for the vector bin packing problem with identical bins. Realistic workloads vary widely across organizations in their heterogeneity and in their resource requirements, so it would be difficult to generalize from any given set of real workloads. Following the work in [3] and [22] we instead run our algorithms on synthetic instances generated randomly from many different distributions, and would like to compare the quality of our solution to the best possible solution. This lets us test our heuristics under a variety of different correlations across dimensions. Our synthetic workloads are taken from previous work on the problem [3] and cover several parameter values, as well as cover positive, negative and no correlations.

Input Distributions We first describe the classes of random instances we use; most of these were also used by [3]. To ease description, we do not scale the bin sizes to be 1 any more, but let the parameter h denote the bin size. The first

six classes all have VM sizes in each dimension drawn randomly and independently from the range $[\alpha, \beta]$ for parameters α and β; thus such a distribution is fully specified by h, α, β. In the first five classes, $h = 1000$, and $[\alpha, \beta]$ is set to $[100, 400], [1, 1000], [200, 800], [50, 200]$, and $[25, 100]$ respectively. Class 6 has $h = 150$ and $[\alpha, \beta] = [20, 100]$; the hardest instances for one-dimensional bin packing in literature are the one dimensional case of this distribution, and this was the motivation for this choice in Caprara and Toth [3].These different parameter value lead to different values for the average number of items per bin.

While in the above classes, the dimensions are independently sampled, classes 7 and 8 have slightly correlated dimensions. Both these classes have $h = 150$ and for each VM, the demand in first dimension v_1 is sampled from $[20, 100]$ as in class 6. Demand in the second dimension v_2 is sampled from $[v_1 - 10, v_1 + 10]$ for class 7 (positive correlation) and from $[110 - v_1, 130 - v_1]$ for class 8 (negative correlation). For higher dimensional variants, dimensions $(2i - 1)$ and $2i$ are correlated as defined, and independent of the other dimensions.

To generate negative correlation across all dimensions, we propose a new distribution. In our class 9, h is set to 100. An item in Class 9 is generated as follows: Throw $2d$ balls independently and randomly into d bins (dimensions), giving us random variables $X_1, \ldots, X_d$ where X_i denotes the number of balls that fall in bin i, and thus the sum of X_i's is $2d$. We choose s uniformly in $[10, 40]$, and set I_i^ℓ to be $sX_i/2$. We further noise the input by adding a uniformly chosen random value in $[0, 1]$ to each dimension. Finally, any items that end up being larger that the bin size in any dimension are ignored. Each item is generated uniformly at random from this distribution.

Thus classes 1-6 are generated from independent distributions, class 7 is positively correlated and classes 8 and 9 are negatively correlated. We remark that [3] considered two other classes that they call 9 and 10, which were very specific to the two-dimensional case and thus we do not show results for those.

Lower Bounds For dimension i, let dem_i be the total demand $\sum_\ell I_i^\ell$ in this dimension. Since each bin has capacity 1 in every dimension, any solution must use at least dem_i bins. We use *SumLB* to denote the quantity $\max_i dem_i$, which is always a lower bound on the optimal solution.

A stronger lower bound can be obtained by writing the (exponential sized) configuration LP where the variables x_C correspond to valid configurations C (i.e. subsets of items that can fit together in one bin), with the constraint that $\sum_{C:i \in C} x_C \geq 1$ for each item i, and we minimize $\sum_C x_C$. Though exponential sized, this linear program can be solved by column generation [9] when the number of items is small. In our experiments, we use the Gurobi [28] LP solver to solve the linear programs, and the Gurobi IP solver to solve the knapsack subroutines that need to be solved to generate columns. This leads to a better lower bound when the linear program can be quickly solved (in our case for n up to a 100).

For the 2-dimensional version of the problem, we have an even better option. We use eight of the input distributions that were used by Caprara and Toth [3],

who also computed lower bounds for these classes. For our experiments, we use the best available lower bound in each case.

4.1 Results

FFD and the New Heuristics We coded up our algorithms in C# and ran them on random instances generated from these classes with several values if n between 100 and 5000. We vary the number of dimensions in the set of values $(2, 3, 4, 6, 10)$. We sample 100 instances from each class of random inputs and compare the average number of bins used by each of the heuristics. For all algorithms involving weights, we chose a_i to be $\exp(0.01 \cdot avdem_i)$.

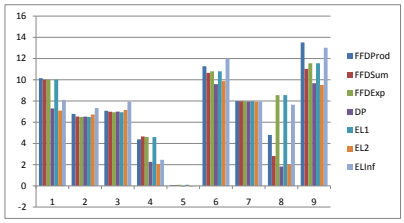

Fig. 1. Performance of the different algorithms with 100 items with 2 dimensions.

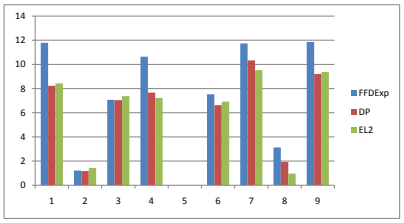

Fig. 2. Performance of the different algorithms with 100 items with 3 dimensions.

Figure 1 shows the percentage overhead (over the configuration LP lower-bound) of several algorithms for various input classes with $n = 100$ in 2 dimensions. These examples show that the algorithms *FFDExpSum*, *DotProduct* and *L2* show the most promise amongst these and we restrict our attention to these. In this and other cases, *FFDAvgSum*'s behavious is very similar to *FFDExpSum* and we omit it from the other experiments. The *LPBasedFFD* and the *DotProductBubble* are visited later in this section.

Already in the two-dimensional case, one can observe that the new heuristics *DotProduct* and *L2* outperform the best of the FFD based heuristics on input classes 6, 7, and 9. Figures 2, 3 and 4 compare these three chosen algorithms in higher dimensions for the input classes described above. For input classes 1, 4, and 5, where the average number of items per bin is large, we see that the geometric heuristics outperform FFD by nearly 5 percent in dimension 6. For other input classes, we see a few percent improvement over FFD. These results are relatively robust to change in dimension and in n. In the appendix, we show the results for $n = 3000$ (compared to the weaker *SumLB*). We also show the effect of increase in the number of dimensions in Figure 5 (and some more in the appendix). We conclude that the new geometric heuristics *L2* and *DotProduct* nearly always do as well as, and often noticeably outperform FFD-based heuristics.

Bubble Search We implemented a bubblesearch version of the *DotProduct* algorithm. We ran it with `nruns` set to 200 and `p` set to 0.6. In Table 1, we compare performance of *DPBubble* and *DotProduct* over different options. We see that

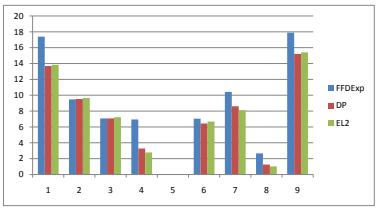

Fig. 3. Performance of the different algorithms with 100 items with 4 dimensions.

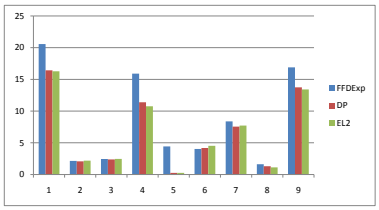

Fig. 4. Performance of the different algorithms with 100 items with 6 dimensions.

bubble search offers some improvement over the simple dot product algorithm at the cost of a higher run time. We observe that the bubble search is easily parallelized. Similar gains were shown for Grasp[3] version of the algorithm.

Input	DP (d=2)	DPBubble(d=2)	%age saving(d=2)	DP (d=6)	DPBubble(d=6)	%age saving(d=6)
1	261.838	260.54	0.495726365	284.05	283.09	0.337968667
2	544.199	543.37	0.152333981	785.142	781.81	0.424381832
3	543.608	544.89	-0.235831702	782.528	780.64	0.241269322
4	128.846	128.3	0.4237617	133.594	133.15	0.332350255
5	64.002	63.75	0.393737696	65.324	65.14	0.281672892
6	420.118	420.03	0.020946496	483.24	481.99	0.25867064
7	408.798	409.08	-0.068982725	448.258	447.41	0.189176769
8	500.506	500.42	0.017182611	501.248	500.11	0.227033325
9	262.204	261.55	0.249424113	282.012	282.53	-0.183680127

Table 1. Number of bins used by DP Bubble and DP

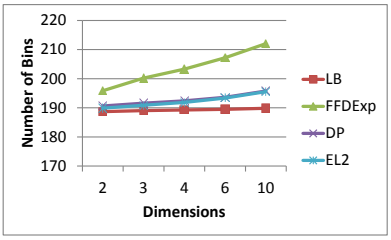

Fig. 5. Performance of the different algorithms with 3000 items with increasing number of dimensions with input class 5.

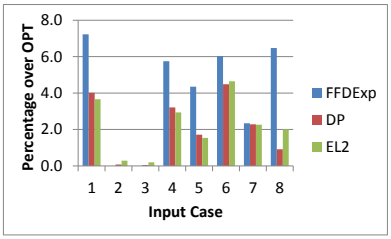

Fig. 6. Performance of the different algorithms as compared to the optimum with 200 items with two dimensions.

Comparison to the Optimal In the two-dimensional case, Caprara and Toth [3] use the distributions 1-8 above, and computed the optimal value of the solution for small values of n. We use the numbers reported in their work to compare ourselves to the optimal rather than to the näive lower bound. Figure 6 shows the percentage overhead of the different algorithms over the optimum taken from Caprara and Toth in the first 8 classes with $n = 200$ in 2 dimensions. We observe that our geometric heuristics perform extremely well, and get within a few percent of the optimum on nearly all the distributions.

Running Times Table 2 shows the running time in seconds of the different algorithms with 1000 items in 6 dimensions generated using input case 5. The running times for the other input classes were not significantly different. All these

FFDSum	FFDExp	DP	EL2	Dpbubble
0.002738433	0.002750367	0.025017907	0.024320518	5.00201

Table 2. Running times in seconds of different heuristics with 1000 items in 6 dimensions; input class 5.

numbers are on an HP Desktop with a Intel Core 2 Duo CPU E8500 @3.16 GHz. The computer has 4GB of memory and runs Windows 7.

5 Conclusions

Our evaulation suggests that our new heuristics compare favorably with natural generalizations of FFD proposed in literature. We show that while the benefit is small when the dimensions are positively correlated (in which case they all reduce to the one dimensional case), the benefit increases when the dimensions are negatively correlated. We believe that the new proposed heuristics should be the algorithms of choice in many applications.

Acknowledgement

We are grateful to Venugopalan Ramasubramanian and Sangmin Lee for several helpful discussions relating to this work, and for pointing us to several related works.

References

1. P. K. Agarwal and J. Erickson. Geometric range searching and its relatives. In *Advances in Discrete and Computational Geometry*, pages 1–56. American Mathematical Society, 1999.
2. N. Bansal, A. Caprara, and M. Sviridenko. Improved approximation algorithms for multidimensional bin packing problems. In *FOCS '06: Proceedings of the 47th Annual IEEE Symposium on Foundations of Computer Science*, pages 697–708, Washington, DC, USA, 2006. IEEE Computer Society.
3. A. Caprara and P. Toth. Lower bounds and algorithms for the 2-dimensional vector packing problem. *Discrete Applied Mathematics*, 111(3):231–262, 2001.
4. J. S. Chase, D. C. Anderson, P. N. Thakar, A. M. Vahdat, and R. P. Doyle. Managing energy and server resources in hosting centers. In *Proc. of ACM Symposium on Operating Systems Principles (SOSP)*, Banff, Canada, Oct. 2001.
5. C. Chekuri and S. Khanna. On multi-dimensional packing problems. In *ACM-SIAM Symposium on Discrete Algorithms*, pages 185–194, 1999.
6. G. Chen, W. He, J. Liu, S. Nath, L. Rigas, L. Xiao, and F. Zhao. Energy-aware server provisioning and load dispatching for connection-intensive internet services. In *Proc. of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, San Francisco CA, Apr. 2008.
7. Y. Chen, A. Das, W. Qin, A. Sivasubramaniam, Q. Wang, and N. Gautam. Managing server energy and operational costs in hosting centers. In *Proc. of the ACM SIGMETRICS Conference*, Banff, Canada, june 2005.
8. W. F. de la Vega and G. S. Lueker. Bin packing can be solved within $(1 + \epsilon)$ in linear time. *Combinatorica*, (1):349–355, 1981.
9. G. Desaulniers, J. Desrosiers, and M. M. Solomon, editors. *Column Generation*. Springer, 2010.

10. T. Feo and M. Resende. Greedy randomized adaptive search procedures. *J. of Global Optimization*, 6:109–133, 1995.
11. M. R. Garey, R. L. Graham, D. S. Johnson, and A. C. Yao. Resource constrained scheduling as generalized bin packing. *Journal of combinatorial theory Series A*, 21:257–298, 1976.
12. B. T. Han, G. Diehr, and J. S. Cook. Multiple-type, two-dimensional bin packing problems: applications and algorithms. *Annals of Operations Research*, (50):239–261, 1994.
13. D. S. Hochbaum, editor. *Approximation algorithms for NP-hard problems*. PWS Publishing Co., Boston, MA, USA, 1997.
14. H. Kellerer, U. Pferschy, and D. Pisinger, editors. *Knapsack Problems*. Springer, 2004.
15. L. T. Kou and G. Markowsky. Multidimensional bin packing algorithms. *IBM J. Res. Dev.*, 21:443–448, September 1977.
16. N. Lesh and M. Mitzenmacher. Bubblesearch: a simple heuristic for improving priority-based greedy algorithms. *Information Processeding Letters*, 97(4):161–169, 2006.
17. K. Maruyama, S. K. Chang, and D. T. Tang. A general packing algorithm for multidimensional resource requirements. *International Journal of Computer and Information Sciences*, 6(2):131–149, 1977.
18. E. Pinheiro, R. Bianchini, E. V. Carrera, and T. Heath. Load balancing and unbalancing for power and performance in cluster-based systems. In *Proc. of the Workshop on Compilers and Operating Systems for Low Power (COLP)*, Barcelona, Spain, Sept. 2001.
19. M. Resende and C. Ribeiro. Greedy randomized adaptive search procedures. In F. Glover and G. Kochenberger, editors, *Handbook of Metaheuristics*, pages 219–249. Kluwer Academic Publishers, 2003.
20. F. C. R. Spieksma. A branch-and-bound algorithm for the two-dimensional vector packing problem. *Computers and Operations Research*, (21):19–25, 1994.
21. S. Srikantaiah, A. Kansal, and F. Zhao. Energy aware consolidation for cloud computing. In *Proc. of the Workshop on Power Aware Computing and Systems (HotPower)*, San Diego, CA, Dec. 2008.
22. M. Stillwell, D. Schanzenbach, F. Vivien, and H. Casanova. Resource allocation algorithms for virtualized service hosting platforms. *Journal of Parallel and Distributed Computing*, 70:962–974, 2010.
23. C. Tang, M. Steinder, M. Spreitzer, and G. Pacifici. A scalable application placement controller for enterprise data centers. In *Proc. of the International Conference on World Wide Web (WWW)*, Banff, Canada, May 2007.
24. V. V. Vazirani. *Approximation algorithms*. Springer-Verlag New York, Inc., New York, NY, USA, 2001.
25. G. J. Woeginger. There is no asymptotic ptas for two-dimensional vector packing. *Inf. Process. Lett.*, 64(6):293–297, 1997.
26. T. Wood, P. J. Shenoy, A. Venkataramani, and M. S. Yousif. Black-box and gray-box strategies for virtual machine migration. In *Proc. of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Cambridge, MA, Apr. 2007.
27. A. C. C. Yao. New algorithms for bin packing. *Communications of the ACM*, (30):540, 1987.
28. Gurobi optimization. `http://www.gurobi.com/`.
29. Microsoft Systems Center Virtual Machine Manager. `http://www.microsoft.com/systemcenter/virtualmachinemanager`.

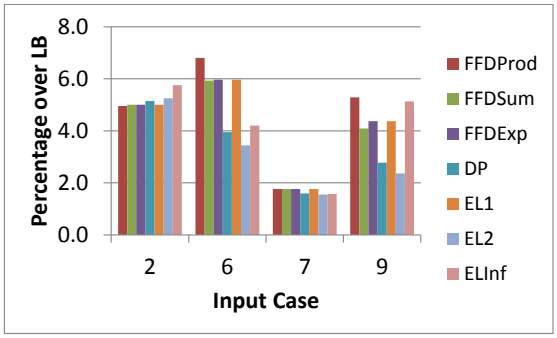

Fig. 7. Performance of the different algorithms with 3000 items with 2 dimensions.

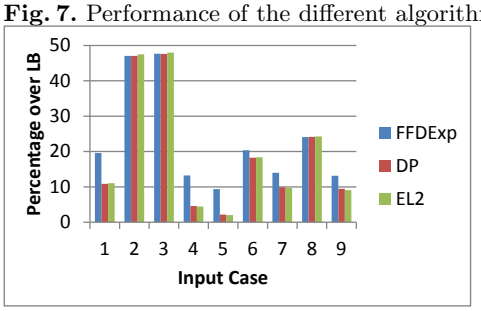

Fig. 8. Performance of the different algorithms with 3000 items with 6 dimensions.

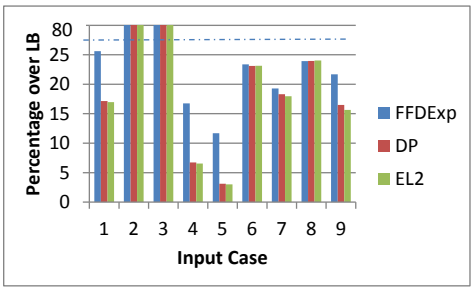

Fig. 9. Performance of the different algorithms with 3000 items with 10 dimensions.

A More Results

FFD and the new heuristics

Figure 7 shows the percentage overhead of several different algorithms in four of the input classes with $n = 3000$ in 2 dimensions. Already in the two-dimensional case, one can observe that the new heuristics *DotProduct* and *L2* outperform the best of the FFD based heuristics on input classes 6, 7, and 9. Figures 8 and 9 compare these three chosen algorithms in 6 and 10 dimensions respectively, for all the nine input classes described above. First note that on input classes 2 and 3, the performance overhead of all the algorithms is fairly large. We believe that this is due to the *SumLB* being inadequate. In both these cases, the item sizes are fairly large and in high dimensions, few pairs of items would be able to occupy the same bin. All of these algorithms would greedily search for pair of items that fit together, and these algorithms place less than 1.1 item per bin on an average in the 10-dimensional case. The *SumLB* only implies an average occupancy of two.

For input classes 1, 4, and 5, where the average number of items per bin is larger, we see that the geometric heuristics outperform FFD by 7-10 percent.

For other input classes, we see a few percent improvement over FFD, except for input class 8.

These results are relatively robust to change in dimension. We show in Figures 10 and 11 the effect of increasing the number of dimensions on the number of bins used with $n = 3000$ with classes 7 and 9.

We conclude that the performance gain of the new geometric heuristics *L2* and *DotProduct* holds for large n as well.

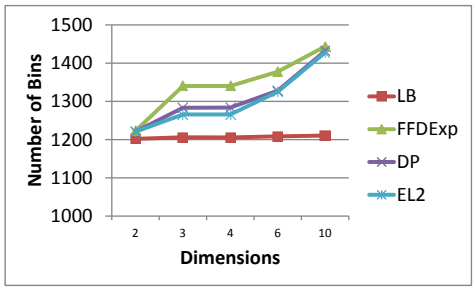

Fig. 10. Performance of the different algorithms with 3000 items with increasing number of dimensions with input class 7.

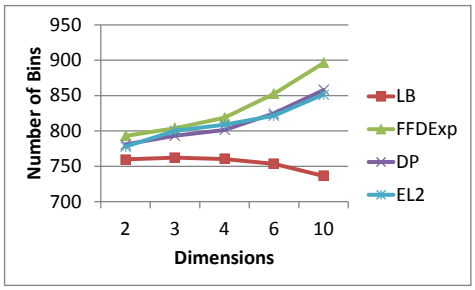

Fig. 11. Performance of the different algorithms with 3000 items with increasing number of dimensions with input class 9.