2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)

Novel Vector Packing Heuristic for VM Placement based on the Divide-and-Conquer Paradigm

SHRAVAN S K
Indian Institute of Science
Bengaluru, India
shravank@iisc.ac.in

J. LAKSHMI
Indian Institute of Science
Bengaluru, India
jlakshmi@iisc.ac.in

***Abstract*—VM Placement algorithms play a crucial role in determining data center utilisation and power consumption. The problem of VM Placement is to obtain an optimal packing of VMs on hosts such that the number of hosts required is minimum. The problem being NP-Hard, it becomes practically infeasible to get an optimal placement within the time constraints for making scheduling decisions. VM Placement can be modeled as a Multi-dimensional Vector Packing Problem(MDVPP). VPSolver, using arc-flow formulation with graph compression, gives an optimal solution for Bin-Packing and related problems. This paper proposes heuristics for large instances of MDVPP, based on the Divide-and-Conquer paradigm, using VPSolver. An extensive evaluation, with 3260 instances, comparing our heuristic with existing popular heuristics in 2D and 3D vector spaces is done which also includes VM Sizes obtained from utilisation traces of a private cloud. It is observed that for most large instances, our heuristic gives better solutions compared to existing methods sometimes at the cost of higher computation time taken. The proposed heuristic gives solutions where the number of bins required is reduced up to 7.34% and 8.15% for 2D and 3D vector spaces respectively.**

***Index Terms*—Vector Bin Packing Heuristic, MDVPP, VM Placement, Consolidation, Resource Management**

I. Introduction

Virtualisation enabled better server consolidation by allowing various instances/workloads to share the same hardware thus improving utilisation compared to stand-alone servers running a single workload. This is a key-enabler for cloud-based technologies. Although with virtualisation utilisation improved considerably compared to the stand-alone servers, many data centers still suffer from poor utilisations leading to idle resources and loss in revenue. Hence, it becomes very important to use data center resources optimally.

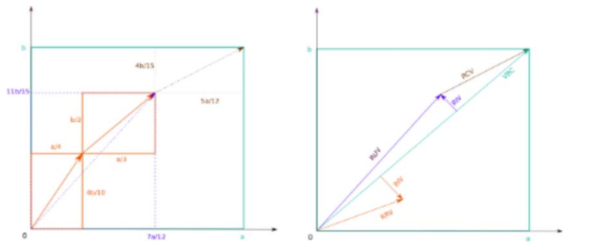
Most research till date, aiming towards efficient use of data center resources, focuses on VM placement algorithms as a means to achieve it. There are numerous studies reported in the literature[11] addressing these diverse characteristics, requirements and optimization goals of data centers through placement algorithms/heuristics. Many objective functions have been defined based on these goals. Single VM placement algorithm could be trying to achieve multiple objectives[11][15][14]. Objectives could be to minimise energy footprint, improve workloads' QoS, minimise network traffic etc. This classification of objectives is not exclusive i.e. a single VM placement algorithm could try to achieve more than one objectives[11].

VM placements are done based on matchmaking of VM requested resources, like CPU cores or network bandwidth, to what is available in any host in the data center. When only one resource is considered for making placement decisions, the VM placement problem is modeled as a 1D (one-dimensional) Bin Packing problem. The dimension here refers to the resource being considered while match-making. When more dimensions are considered, the VM Placement problem becomes a *Multi-dimensional Vector Packing Problem(MDVPP)*[8] or *Vector Bin Packing Problem* or just *Vector Packing Problem*. As the problem of VM Placement is NP-Hard(time taken to arrive at an optimal solution is too long for larger instances) we propose a grouping-based heuristic to solve the VM Placement problem which gives near-optimal solutions in time scales suitable for VM Placement scenarios.

II. Vector Bin Packing Problem

The vector bin packing problem(VBP) or MDVPP is a generalization of bin packing with multiple constraints. In this problem we are required to pack items, represented by p-dimensional vectors, into as few bins as possible. By means of reductions to vector packing, several cutting and packing problems, including the one-dimensional bin packing and cutting stock problems, can be solved. Figure 1 shows a Vector Bin Packing instance and the terminology that is used. $\vec{W} =< a, b >$ is the Vector Bin Capacity(VBC) and $\vec{v_1} =< a/4,\ 4b/10 >$ and $\vec{v_2} =< a/3,\ b/3 >$ are the Item Vectors(Resource Request Vector - RRV). Item vectors $\vec{v_1}, \vec{v_2}$ are placed in the bin $\vec{W}$. $\vec{v_1} + \vec{v_2} =< 7a/12,\ 11b/15 >$ represents the current occupied portion of the Vector Bin(Resource Utilisation Vector - RUV). The remaining capacity of the Vector Bin(RCV) is, $\vec{W} - (\vec{v_1} + \vec{v_2})\ =\ < 5a/12,\ 4b/15 >$. Resource Imbalance Vector(RIV) is calculated as the vector difference between RUV's projection on the VBC and the RUV. Similarly, RIV of an item vector is defined as the vector difference between RRV's projection on VBC and the RRV.

Many exact methods have been proposed to solve VBP which is NP-Hard. Among the exact methods to solve Bin Packing and Related problems, Filipe Brandão's arc flow-formulation with graph compression method[1] has solved some of the previously unsolved benchmark instances and



Example 1 : $\vec{W} =< a, b >$ is the Vector Bin Capacity. $\vec{v_1} =< a/4,\ 4b/10 >$ and $\vec{v_2} =< a/3,\ b/3 >$ are the Item Vectors placed in the Vector Bin.

Fig. 1: A 2D Vector Bin Packing Example

performs very well compared to other existing methods[3][5]. These methods solved for optimal solutions of only smaller instances of the Vector Bin Packing problem. As optimal solutions for larger instances takes too long to arrive at, we propose heuristics which give near-optimal solutions for larger instances of the Vector Bin Packing Problem. The proposed heuristic is evaluated with 2D and 3D vector spaces and also with VM Sizes collected from utilisation traces of a private cloud. Other heuristics are also proposed to solve the Vector Bin Packing problem[7][8][4][6].

Data centers typically have multiple VMs of same configurations. Such scenarios are well represented and solved easily using a CSP(Cutting Stock Problem) formulation. We make use of [1]'s arc-flow formulation and graph compression, which uses a CSP formulation, to build our heuristic. This formulation is explained in the following section.

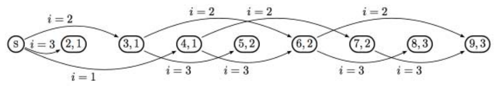
III. Arc-flow formulation and graph compression

Among other methods to solve Bin Packing exactly, one of the most important ones is Valério de Carvalho's arc-flow formulation with side constraints[2]. This model has a set of flow conservation constraints and a set of demand constraints to ensure that the demand of every item is satisfied. Filipe Brandão's work extends and generalizes this arc-flow formulation which solves other variants of the Bin Packing problem, like Cutting Stock Problem(CSP), cardinality-constrained bin packing and two- constraint bin packing(2-CBP) and propose a graph compression method to reduce the graph size dramatically thus making it possible to solve even harder problems[1]. Filipe Brandão arc-flow formulation with graph compression are discussed next.

A. Filipe Brandão's arc-flow formulation with graph compression

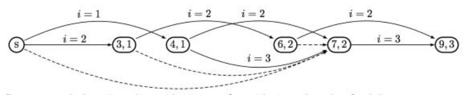
Filipe Brandão[1] proposes a generalization of Valério de Carvalho's model, still based on representing the packing patterns by means of flow in a graph, and assigns a more general meaning to vertices and arcs. This model generates graph of the flow problem taking into considerations the generalisations and enhancements to Valério de Carvalho's model. This graph is further optimised to reduce the graph size using a 3-step graph compression algorithm[1] with no valid packing patterns being removed. The compressed graph will be solved by a general-purpose mixed-integer optimization solver and then the solution to the problem instance is extracted. This work uses VPSolver[9] which is based on this arc-flow formulation with graph compression.

Figure 2a shows the graph associated with a cardinality constrained bin packing instance with bins of capacity 9 and cardinality limit 3 i.e. a bin can hold up to 3 items$\{W = (9, 3)\}$ and items of sizes 4, 3, 2$\{w_1 = (4, 1), w_2 = (3, 1), w_3 = (2, 1)\}$ with demands 1, 3, 1$\{b_1 = 1, b_2 = 3, b_3 = 1\}$, respectively. Figure 2b gives the corresponding compressed graph after the 3-step graph compression algorithm.



Graph corresponding to a cardinality constrained bin packing instance with bins of capacity $W = 9$, cardinality limit 3 and items of sizes 4, 3, 2 with demands 1, 3, 1, respectively.

(a) Step-1 Graph



The Step-4 graph has 7 nodes and 15 arcs (considering also the final loss arcs connecting internal nodes to T). In this case, the only difference from the Step-3 graph is the node (5, 2) that collapsed with the node (4, 1). The initial Step-1 graph had 9 nodes and 18 arcs.

(b) Step-4 Graph

Example 2 : $W = (9, 3), w = \{(4, 1), (3, 1), (2, 1)\}, b = \{1, 3, 1\}$

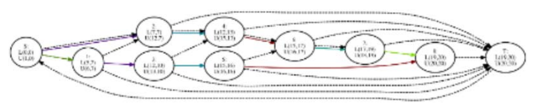
Fig. 2: Initial Step-1 Graph and the Step-4 graph after the 3-step graph compression algorithm for Example 2

The proposed heuristic uses VPSolver[9] which is based on this formulation and graph compression. So, to understand the computation time of the novel heuristic, it is important to understand the dependencies of this formulation and graph compression method. The factors affecting the graph size, which in turn affects the computation time of the algorithm, are explained next.

B. Computation Time Dependencies

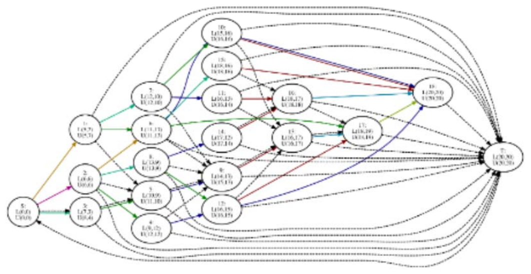
Computation time is directly proportional to the graph size (number of arcs and number of vertices) obtained after graph compression. How the graph size varies based on the values of W, w, b and the factors affecting these are explained as follows:

Number of items: As discussed earlier, the arcs of the graph correspond to items. Intuitively, it can be said that higher the number of items, larger the graph size(Step-1 graph) would be. What matters finally is the graph size obtained after the graph compression step. So, even after the number of items is increased, as a result of the graph compression step, the increase in the Step-4 graph size may not be higher. Although generally the graph size increases with the number of items, a conclusion stating that 'an increase in the number of items results in an increase in the graph size' is not valid for all cases.



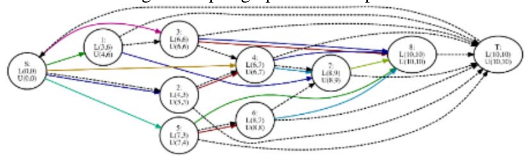
Example 3 : $W = (20, 20), w = \{(1, 4), (2, 1), (5, 7), (3, 6), (2, 2), (7, 3)\}$, $b = \{1, 1, 1, 1, 1\}$

Fig. 3: Step-4 graph for Example 3



Example 4 : $W = (20, 20)$,
$w = \{(1, 4), (2, 1), (5, 7), (3, 6), (2, 2), (7, 3), (4, 3), (6, 6)\}$,
$b = \{1, 1, 1, 1, 1, 1, 1, 1\}$

Fig. 4: Step-4 graph for Example 4



Example 5 : $W = (10, 10)$, $w = \{(1, 4), (2, 1), (5, 7), (3, 6), (2, 2), (7, 3), (4, 3), (6, 6)\}$, $b = \{1, 1, 1, 1, 1, 1, 1, 1\}$

Fig. 5: Step-4 graph for Example 5

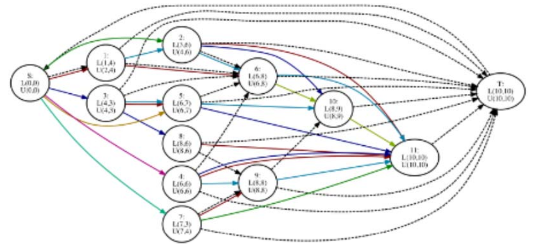
Figure 3 shows the graph for Example 3. Each colour of the arcs represents an item and the dotted arcs are loss arcs as explained in the previous section. Two items (4, 3) and (6, 6) are added to *w* resulting in the Example 4 in Figure 4.

From Figures 3 and 4, it can be seen that the graph size depends on the number of items.

Items' distribution with respect to the Bin Capacity: It can be observed that the items' sizes(distribution) with respect to the bin size also influences the size of the graph. The smaller the items are with respect to the bin size, the larger the graph could be as there would be more intermittent vertices and arcs for smaller items which results in longer path lengths from 0 to W. The larger the items are, the shorter the path lengths will be from 0 to W. The bin capacity in Example 4 is reduced from $W = (20, 20)$ (Figure 4) to $W = (10, 10)$ keeping all other values the same to obtain the graph corresponding to Example 5 in Figure. 5. Reduction in the graph size can be noticed when the bin size was reduced.

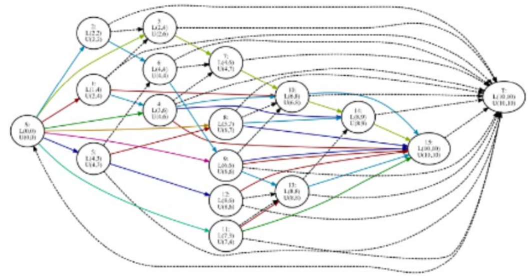
Demands of items: Each item can have demands equal to or greater than 1. When items have demands greater than 1, these also add up to the nodes and arcs of the graph depending on the bin capacity. Items in Figure 5 had demands of 1 each. We increase the demands of items to 2, 5 and 10 to obtain the graphs in Figures 6, 7 and 8 respectively. The graph size increased when the demands were increased to 2 and 5 but the graph is same for demands of 5 and 10. This is because after a certain limit, no additional vertices are added and thus any further increase in demands has no effect on the graph size. Hence, the graph size also depends on the demands of the items.

Number of Dimensions: By increasing dimension of the problem space, the value space of the graph's nodes also increases thus resulting in more number of nodes and arcs. We add a dimension to Example 4 (Figure 4) to obtain the graph



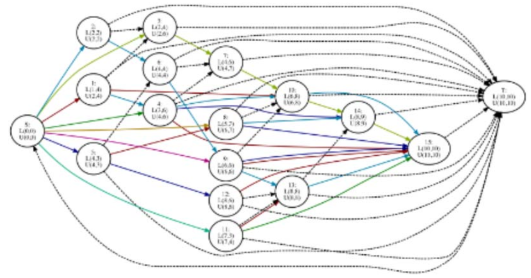
Example 6 : $W = (10, 10)$, $w = \{(1, 4), (2, 1), (5, 7), (3, 6), (2, 2), (7, 3), (4, 3), (6, 6)\}$, $b = \{2, 2, 2, 2, 2, 2, 2, 2\}$

Fig. 6: Step-4 graph for Example 6



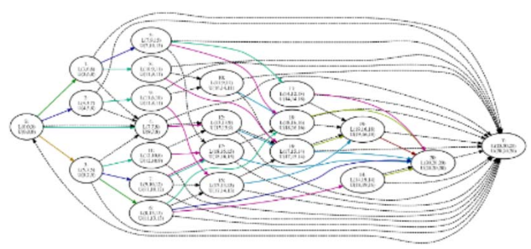
Example 7 : $W = (10, 10)$, $w = \{(1, 4), (2, 1), (5, 7), (3, 6), (2, 2), (7, 3), (4, 3), (6, 6)\}$, $b = \{5, 5, 5, 5, 5, 5, 5, 5\}$

Fig. 7: Step-4 graph for Example 7



Example 8 : $W = (10, 10)$, $w = \{(1, 4), (2, 1), (5, 7), (3, 6), (2, 2), (7, 3), (4, 3), (6, 6)\}$, $b = \{10, 10, 10, 10, 10, 10, 10, 10\}$

Fig. 8: Step-4 graph for Example 8



Example 9 : $W = (20, 20, 20)$, $w = \{(1, 4, 2), (2, 1, 4), (5, 7, 5), (3, 6, 8), (2, 2, 5), (7, 3, 3), (4, 3, 7), (6, 6, 1)\}$, $b = \{1, 1, 1, 1, 1, 1, 1, 1\}$

Fig. 9: Step-4 graph for Example 9

Number of Items	100	200	300	400	500
Time Taken(secs)	1.83	17.3	180.0	797.8	2520.1

Increase in computation time taken with increase in number of item types. The 2D class of instance : Items' range for both dimensions = [10, 60], bin size = [100, 100]. The results are averaged for 10 randomly generated instances.

TABLE I: Average computation time taken - Comparison

in Figure 9. It can be seen that the number of dimensions also plays a role in determining the graph size. Increasing the number of dimensions may potentially result in a larger graph size.

IV. Novel Grouping-Based Heuristic

As Vector Packing is NP-Hard, it becomes infeasible to get optimal solutions within the time constraints to make VM placement decisions for large inputs. In this section, we describe a novel heuristic to get near-optimal solutions at lesser time intervals.

Divide and Conquer: In the previous section, we observed that increasing the number of items increases the graph size and in turn the computation time required. Table I shows how computation time taken increases, to obtain optimal solution using VPSolver, with increasing number of items. It can be observed that for large instances the time taken to arrive at the optimal solution will be very large and often impractical. By dividing the problem space into smaller sub-problems a near-optimal solution can be arrived at in lesser time(within a decision making interval). For example, an instance having 500 items can be divided into 5 smaller groups of 100 items each and can be solved independently to get a sub-optimal solution in lesser time than what it takes to compute the optimal solution. How close such a solution is to the optimal one and the computation time required for solving the smaller groups depends on how the items are divided into groups. This heuristic is described in Algorithm 1. $GROUPING\_ALGO(w, b, m)$ divides the items into groups based on some criteria. Each of these groups is solved by a VPSolver instance. These grouping techniques are discussed next.

Algorithm 1: Grouping based heuristic

```
Input  : Items $w$, Demands $b$, Bin Capacity $W$, Number of groups $m$
Output: Number of bins required
begin
    $item\_grps = [[\ ], [\ ], ...[\ ]]$
    $demand\_grps = [[\ ], [\ ], ...[\ ]]$
    $item\_grps, demand\_grps =$
      $GROUPING\_ALGO(w, b, m)$
    $i = 0, bins = [\ ]$
    while  $i < m$ do
        $bins[i] =$
          $VPSolver(item\_grps[i], demand\_grps[i], W)$
        $i = i + 1$
    return $sum(bins)$
```

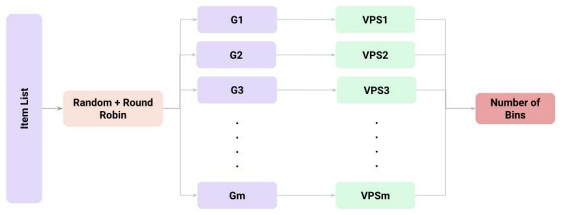


Fig. 10: Using *Random + Round Robin* heuristic

A. Random Grouping

A very straight-forward or brute-force approach to divide the items into smaller groups is to choose an element at random and allot the item to a group in a Round Robin fashion. And then each of the groups can be solved independently by a VPSolver instance as shown in Figure 10. $G1, G2, G3, ..., Gm$ are the groups which are obtained after dividing the input using Random + Round Robin algorithm. Each of $G1, G2, G3, ..., Gm$ is given as an input to $VPS1, VPS2, VPS3, ..., VPSm$, these are independent of each other and can execute in parallel. The combined number of bins of each VPSolver instance is the solution of the heuristic. Such a non-uniform grouping may lead to higher computation time taken and the obtained solution can be far from optimal.

Counter Examples for Random Grouping: Consider a simple example in the 1D space - there are n items to be packed into bins of capacity x. $n/2$ items have weights $> 2x/3$ (uniformly distributed) and the other $n/2$ items have weights $< x/3$(uniformly distributed). And suppose the number of groups the items are to be divided into is 2 and all the $n/2$ items having weights $> 2x/3$ are in one group and the rest $n/2$ items having weights $< x/3$ in the other, then the solution will be $n/2$ bins for the first group as every item in this group has weight $> 2x/3$ only 1 item can be placed in each bin of capacity x. The second group will have an upper bound of $n/6$ bins as each item has weight $< x/3$ a maximum of 3 such items can be placed in a bin of capacity x, as the items are uniformly distributed the bins required for the second group is close to $n/12$. The bins required for both these groups is close to $n/2 + n/12$ whereas the optimal solution is close to $n/2$ bins. Also, the group having smaller items takes longer to compute due to the larger graph size (as discussed in previous section). Although this is an extreme example of grouping, the point to be noted is that using random grouping could lead to worse solutions with higher computation time required.

B. Vector Group Sort

Grouping technique plays an important role in determining (i) how close the computed solution is to the optimal and (ii) time taken to compute. As seen with Random grouping, non-uniform distribution of groups could lead to solutions being away from optimal and may result in higher computation time. In order to make the grouping result in solutions near to optimal, the groups should have items such that the distribution of items in groups and that in the input are similar. Here, we propose a technique to group items such that the items

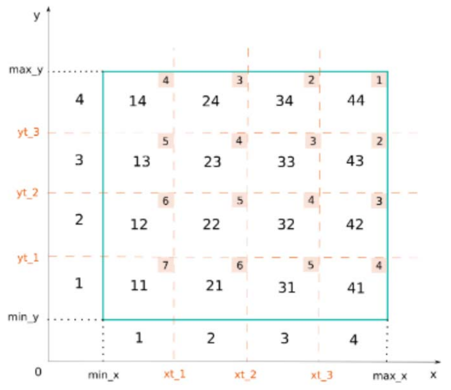


Fig. 11: Example grouping for a 2D vector space

are distributed uniformly across the groups and further in the results the effectiveness of this technique is shown by comparing it with random grouping and other heuristics in this space.

Consider a scenario where the items and bins are one dimensional. To group items such that their item distribution represents that of the initial input set, the given items can be sorted (ascending/descending) and then distributed into groups in a Round-Robin fashion. Sorting multi-dimensional vectors is not straight-forward - whereas in 1D, there is only 1 value for an item to be compared. The idea behind Vector Group Sort is that the entire vector space bound by the input items(vectors) is partitioned into smaller regions/spaces called threshold regions. Items in each such threshold region are distributed into groups in a Round-Robin fashion. Every such group is solved by a VPSolver instance. This is explained in detail below.

Figure 11 shows an example grouping for 2D instances. *min_x, max_x* and *min_y, max_y* represent the range of values of the items along dimensions *x* and *y* respectively. Each dimension has a set of threshold values, dimension *x* has thresholds as *xt_1, xt_2 and xt_3* and dimension y has thresholds as *yt_1, yt_2 and yt_3*. Both *min_x, max_x* and *min_y, max_y* are bounds of dimensions and can also be considered as thresholds of that dimension. The number of thresholds for each dimension and their values can be based on the level of granularity of fragmentation required and may also consider the item distribution within the vector space, say for example the threshold values of a dimension can be decided based on average, standard deviation etc., of the values of that dimension. The fragmented vector spaces formed by the intersections of threshold values of all dimensions are called *threshold regions* - there are 16 such threshold regions(11, 12,...,43, 44) formed by threshold values - *min_x, xt_1, xt_2, xt_3, max_x* and *min_y, yt_1, yt_2, yt_3, max_y*. Each threshold region can be given a value based on its location with respect to the threshold values of all dimensions - this value denotes how far the region is from the origin. In the example, we can see that the threshold region numbered as *44* is constituted by region 4 on dimension *x* (region between *xt_3* and *max_x* and by region

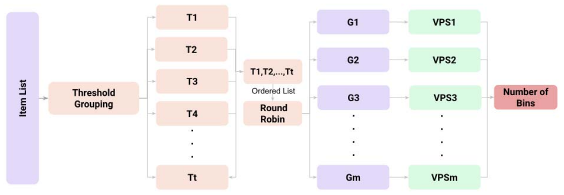


Fig. 12: Using *Vector Group Sort + Round Robin* heuristic

4 on dimension *y* ((region between *yt_3* and *max_y*) - therefore the *threshold region number* can be 4+4=8(not shown in the Figure). Threshold regions numbered 34 and 43 are constituted by regions 3, 4 and 4, 3 respectively(3+4=4+3=7) and similarly for other threshold regions. Each threshold region is given a *threshold region label* based on its *threshold region number*. Starting from the threshold region with the highest *threshold region number* i.e. region 44 is given *threshold region label* as 1 and regions 34 and 43 are labeled 2 and so on. All the threshold regions with the same *threshold region label* are clubbed to form *threshold groups*. In the example Figure there are 7 *threshold groups*.

The Vector Group Sort heuristic checks each item and according to its position in the vector space and thus its *threshold region label*, places the item into the corresponding *threshold group*. After all the items are distributed into threshold groups, each threshold group may further be ordered. Ordering can be Random, based on magnitude of some or all of the dimensions etc. Then all the threshold groups are appended into a single list in the order of increasing/decreasing threshold region labels. This ordered list is then distributed into groups in a Round-Robin fashion. Each such group is solved by a VPSolver instance and their aggregated solution is the number of bins of this heuristic. The complete algorithm is described using Figure 12 and in Algorithm 2. min_vals and max_vals are the list of minimum and maximum values in each dimension of all the n dimensional items in w. $threshold\_values$ is the list of threshold values in all n dimensions. The function $getThresholdLabel$ returns the *threshold region label* of an item based on its position in the vector space and the threshold values. The function $RRGrouping$ distributes items and their demands in a Round-Robin fashion into m groups.

V. Evaluation Setup

This section describes the evaluation setup to compare the proposed heuristics with existing heuristics with respect to the bins required and the time taken to compute.

As exact methods for large instances are infeasible, many heuristics are proposed to solve 1-D Bin Packing Problem - First-Fit Decreasing(FFD), Best-Fit First, Worst-Fit, Next-Fit etc. For multi-dimensional vector packing, two of the prominent principles of packing are based on Dot Product of the vectors and the second one is based on aligning item vectors(RRV) towards the vector bin capacity(VBC). These algorithms are described below.

Vector Dot : At each instance, this heuristic places the item which has the maximum dot product of the vector of item's requirement and the vector of the remaining capacities of the

Algorithm 2: Vector Group Sort + Round Robin

```
Input  : Items $w$, Demands $b$, Number of groups $m$
Output: Item Groups, Demand Groups
begin
    $min\_vals = [min_1, min_2, ..., min_n]$
    $max\_vals = [max_1, max_2, ..., max_n]$
    $threshold\_values = [$
    $d1 = [td1_1, td1_2, ..., td1_{R1}],$
    $d2 = [td2_1, td2_2, ..., td2_{R2}],$
    .....
    $dn = [rn_1, rn_2, ..., rn_{Rn}]$
    $]$
    $threshold\_groups = [[\ ], [\ ], ...[\ ]]$
    for $item\ in\ w$ do
        $lbl = getThresholdLabel(item)$
        $threshold\_groups[lbl].append(item)$
    $ordered\_list = Ordering(threshold\_groups)$
    $ordered\_demands$
    $item\_grps, demand\_grps =$
      $RRGrouping(ordered\_list, ordered\_demands, m)$

    return $item\_grps, demand\_grps$
```

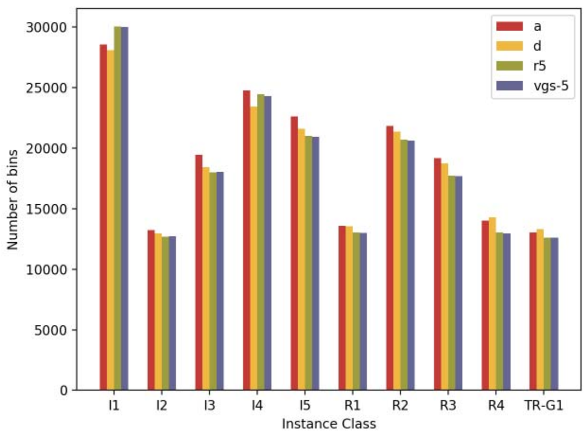
current open bin. Referring to Figure 1, this is the dot product of RRV(Resource Request Vector) of an item vector and the RCV(Remaining Capacity Vector) of the current open bin. The better these two vectors align(the lesser the angle between them), the higher the dot product will be.

Vector Align : At each instance, this heuristic places that item on a host which has the least magnitude of the vector sum of RIV(resource Imbalance Vector) of VM and the RIV of a host's current occupancy(RUV). The least magnitude of the vector sum indicates that a VM with complementary resource requirement, with respect to a host's current occupancy, is placed. Referring to Figure 1, this is the vector sum of RIV of an item vector's RRV and RIV of a bin's RUV.

Instanc Class	Item Dim1 range(u.d.)	Item Dim2 range(u.d.)	Item Dim3 range(u.d.)	Bin Dim
I1	[1, 100]	[1, 100]	[1, 100]	[100], [100], [100]
I2	[10, 40]	[10, 40]	[10, 40]	[100], [100], [100]
I3	[10, 60]	[10, 60]	[10, 60]	[100], [100], [100]
I4	[10, 80]	[10, 80]	[10, 80]	[100], [100], [100]
I5	[20, 60]	[20, 60]	[20, 60]	[100], [100], [100]
I6	[20, 40]	[20, 40]	[20, 40]	[100], [100], [100]
R1	[10, 40]	[dim1*2, dim1*4]	[10, 40]	[100], [300], [100]
R2	[20, 60]	[dim1*2, dim1*4]	[20, 60]	[100], [300], [100]
R3	[20, 50]	[dim1-10, dim1+10]	[20, 50]	[100], [100], [100]
R4	[40, 80]/[120, 160]	[120, 160]/[[40, 80]	[40, 160]	[400], [400], [400]

TABLE II: 2D and 3D Instance classes

To evaluate, different classes of inputs are chosen where each class represents a particular type of distribution of the items. The instance classes for 2D and 3D cases can be seen in Table II respectively - for 2D instances dimensions 1 and 2 are considered. For the instances with a prefix of **I**(Independent) the values chosen for each dimension is independent of the other dimensions whereas instances with a prefix of **R**(Related) have a correlation among the dimensions as specified in the tables. Also, VM Sizes based on utilisation traces collected from a private cloud(custom VMs in [10]) are considered for evaluation. For the 2D vector space, the class of VMs TR-G1 from the traces is used where cpu and memory values are considered. For the 3D vector space, the class of VMs TR-I1 from the traces is used where cpu, memory and disk storage values are considered.



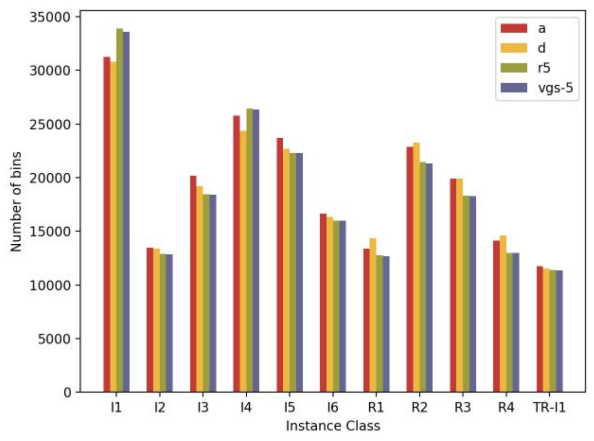
Average Bins required comparison for 2D instances with number of item types 500 and total demands of items = 50000. Terminology : (a=Vector Align, d=Vector Dot, r5=random_RR with 5 groups, vgs-5=vector group sort with 5 groups)

Fig. 13: Bins required - 2D instances

To evaluate, the 2 variants/formulations of the Vector Bin Packing problem are considered - (i) Bin Packing problem and (ii) Cutting Stock problem. The bin packing problem (BPP) can be seen as a special case of the cutting stock problem(CSP). In BPP, n objects of m different weights must be packed into a finite number of bins, each with capacity W, in a way that minimizes the number of bins used. However, in CSP, the items of equal size (which are usually ordered in large quantities) are grouped into orders with a required level of demand; in BPP, the demand for a given size is usually close to one. The different number of item types, w, evaluated are $100, 200, 300, 400\ and\ 500$. The different total demands, Σb, evaluated are $100, 1000, 10000, 200, 2000, 20000, ..., 500, 5000, 50000$. For each instance class with distinct total demands, 10 instances are generated randomly and all the heuristics are evaluated and their averages are presented next. The grouping heuristics are evaluated using VPSolver v3.1.2[12] with IBM ILOG CPLEX Optimisation Studio 12.8[13].

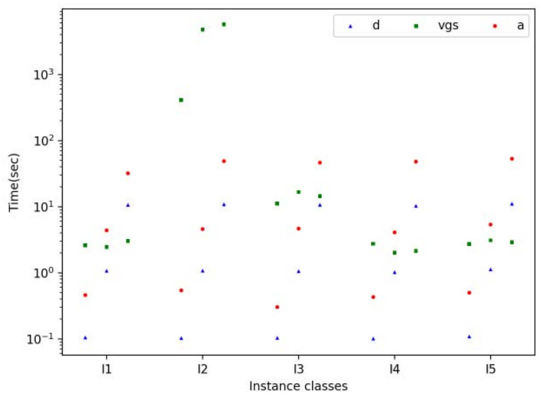
VI. Results

Figures 13 and 14 plot the average bins required for each 2D and 3D instance class types having total demands of

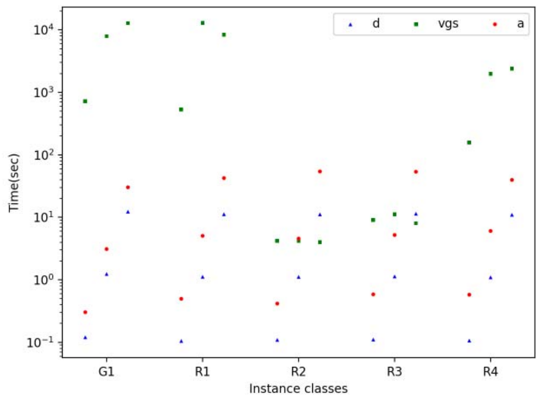


Average Bins required comparison for 3D instances with total number of items = 50000. Terminology : (a=Vector Align, d=Vector Dot, r5=random_RR with 5 groups, vgs-5=vector group sort with 5 groups, TR-I1 is solved with 10 groups)

Fig. 14: Bins required - 3D instances



(a) 2D Instance Classes - I1, I2, I3, I4, I5



(b) 2D Instance Classes - R1, R2, R3, R4, TR-G1

Fig. 15: Total Computation Time taken - 2D instances

Instance Class/Total Item Demands	10K (Optimal)	20K (g=2)	30K (g=3)	40K (g=4)	50K (g=5)
I1	0.508	-3.212	-4.980	-6.736	-6.813
I2	4.165	2.318	2.631	2.219	2.025
I3	5.566	3.989	2.700	2.450	2.126
I4	1.150	-0.888	-1.310	-2.833	-3.715
I5	4.970	4.249	3.870	3.315	2.968
I6	4.711	4.083	3.743	-	-
R1	4.828	4.783	4.577	4.250	4.190
R2	5.538	4.443	3.958	3.609	3.496
R3	6.711	5.738	5.722	5.691	5.527
R4	10.486	8.529	9.028	7.925	7.340
TR-G1	3.709	3.784	3.818	3.263	3.279

(a) 2D instances

Instance Class/Total Item Demands	10K (Optimal)	20K (g=2)	30K (g=3)	40K (g=4)	50K (g=5)
I1	0.509	-3.549	-5.926	-8.424	-9.032
I2	6.014	4.911	4.435	3.825	3.858
I3	8.398	5.968	3.939	4.918	4.250
I4	1.837	-3.533	-4.957	-7.801	-8.199
I5	4.236	3.217	2.816	2.106	1.730
I6	3.319	2.815	2.849	1.982	2.084
R1	7.458	4.871	5.619	4.890	5.045
R2	9.222	7.509	7.658	7.197	6.679
R3	10.349	9.852	8.826	8.074	8.159
R4	11.857	10.372	8.645	8.146	8.004
TR-I1*	2.511	1.888	1.677	1.685	1.618

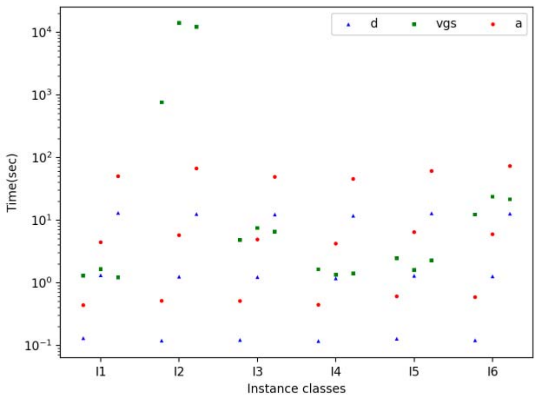
(b) 3D instances

Percentage reduction of bins with Vector Group Sort heuristic compared to the best of Vector Dot and Vector Align for all instances where total number of items - 10000, 20000,....,50000. g=*n* where *n* is the number of groups. *TR-I1 is solved with 2, 4, 6, 8, 10 groups respectively.

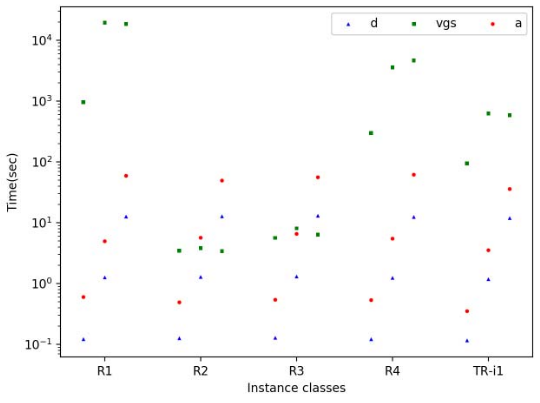
TABLE III: Percentage reduction of bins with Vector Group Sort

50,000 items. It can be seen that the proposed grouping-based heuristics perform better than Vector Dot and Vector Align(except for classes I1 and I4). Classes I1 and I4 have larger item sizes(with respect to the bin capacity - Table , II) Vector Align performs better than Vector Dot in the case of R4 and TR-G1 as the item vectors are complementary in nature(refer Tables II and the principle behind Vector Align exploits this kind of item distribution. Vector Group Sort gives slightly better results compared with random grouping as the distribution with Vector Group Sort is uniform(as discussed in the previous section).

Table III plots the percentage reduction in bins, when Vector Group Sort is used, compared with the best of Vector Dot and Vector Align heuristics. It can be observed that as the number of groups are increased the percentage gain tends to decrease. The proposed heuristic takes upto 7.34% and 8.15% fewer bins, for large 2D and 3D instances respectively, compared to existing heuristics. It can be seen that for instance classes I1 and I4 where the item sizes are larger the gap between the optimal solution and that of existing heuristics is less and hence grouping-based heuristics perform worse with such cases.



(a) 3D Instance Classes - I1, I2, I3, I4, I5, I6



(b) 3D Instance Classes - R1, R2, R3, R4, TR-I1

Fig. 16: Total Computation Time taken - 3D instances

Figures 15, 16 plot average time taken by Vector Dot and Vector Align and average total time taken(sum of time taken by all groups) of Vector Group Sort heuristic. For each instance class, 3 vertical groups of points can be observed - each such vertical group corresponds to 500, 5000, 50000 total demands of items respectively. It can be seen that the time taken by Vector Dot and Vector Align is dependent on the total number of items but the time taken by Vector Group Sort depends more on the instance class type - for classes having smaller item sizes(with respect to the bin capacity) time taken is higher and for classes having larger items time taken is lesser - as explained in Section III-B. The time taken by Vector Group Sort is sometimes lesser and sometimes higher than Vector Dot and Vector Align depending on the instance class. For a given instance, by increasing the number of groups, the computation time required decreases as the group size decreases but the obtained solution may result in higher number of bins required. The solutions obtained could get better with lesser number of groups but at the cost of higher computation time. Also, cases where the item sizes are large(like I1 and I4) optimal solutions can be found in lesser time intervals(as the graph size would be smaller as seen in III-B).

VII. Conclusions

VM Placement policy plays a key role in determining data center utilisation. Arriving at an optimal mapping of VMs to hosts is a challenge for large number of VMs. A novel vector-based grouping heuristic based on the divide-and-conquer approach is proposed using VPSolver which is based on arc-flow formulation with graph compression. Typically data centers have large number of VMs where multiple VMs have same configurations/sizes. Although the total number of VMs could be in tens of thousands, as such scenarios can be solved using a CSP formulation, if the number of VM types/configurations is less(multiples of tens), then optimal VM Placement solutions can be found using VPSolver in the order of a few seconds. Our grouping-based heuristics can be used where the number of VM types are large to obtain near-optimal solutions. Extensive evaluation of 3260 instances which include VM Sizes obtained from utilisation traces of a private cloud shows that for instances with total item demands of 50,000 our grouping-based heuristic takes upto 7.34% and 8.15% fewer bins, for 2D and 3D instances respectively, compared to existing heuristics - this percentage at the scale of the number of hosts in a data center is valuable.

References

[1] Brandao. F, "Bin Packing and Related Problems: Pattern-Based Approaches," *Master's thesis, Faculdade de Ciencias da Universidade do Porto, Portugal* 2012

[2] J. M. Valério de Carvalho, "Exact solution of bin-packing problems using column generation and branch-and-bound," *Annals of Operations Research*, 1999

[3] Results for Bin Packing and Related Problems, https://research.fdabrandao.pt/research/vpsolver/results/

[4] Rina Panigrahy et.al, "Heuristics for Vector Bin Packing", www.microsoft.com/en-us/research/publication/heuristics-for-vector-bin-packing/

[5] M. Delorme et. al, "Bin packing and cutting stock problems: Mathematical models and exact algorithms", in *European Journal of Operations Research* Vol 255 Iss 1 Nov 2016

[6] S. Jaganti et.al, "Scalable and direct vector bin-packing heuristic based on residual resource ratios for virtual machine placement in cloud data centers", in *Computers & Electrical Engg* 2018

[7] A. Singh et.al, "Server-storage virtualization: integration and load balancing in data centers", *ACM/IEEE conference on Supercomputing* 2008

[8] M. Mishra et.al, "On Theory of VM Placement: Anomalies in Existing Methodologies and Their Mitigation Using a Novel Vector Based Approach", *IEEE CLOUD* 2011.

[9] Filipe Brandão', "VPSolver 3: Multiple-choice Vector Packing Solver", arxiv.org/abs/1602.04876

[10] Shravan S K et.al, "Towards Improving Data Center Utilisation by Reducing Fragmentation", *IEEE CLOUD* 2018.

[11] F. L. Pires et.al, "Virtual machine placement literature review", 2015.

[12] "VPSolver 3.1.2", http://github.com/fdabrandao/vpsolver/releases

[13] www.ibm.com/products/ilog-cplex-optimization-studio

[14] F. L. Pires et.al, "Many-Objective Virtual Machine Placement", *Journal of Grid Computing*, 2017.

[15] Jing Xu et. al, "Multi-objective Virtual Machine Placement in Virtualized Data Center Environments", *IEEE International Conference on Green Computing and Communications*, 2014.