Variable size vector bin packing heuristics - Application to the machine reassignment problem

Michaël Gabay, Sofia Zaourar

Variable size vector bin packing heuristics – Application to the machine reassignment problem

Michaël Gabay* Sofia Zaourar†

October 1, 2013

Abstract

In this paper, we introduce a generalization of the vector bin packing problem, where the bins have variable sizes. This generalization can be used to model virtual machine placement problems. In particular, we study the machine reassignment problem. We propose several greedy heuristics for the variable size vector bin packing problem and show that they are flexible and can be adapted to handle additional constraints. We highlight some structural properties of the machine reassignment problem and use them to adapt our heuristics. We present numerical results on both randomly generated instances and Google realistic instances for the machine reassignment problem.

Keywords Variable Size Vector Bin Packing · Heuristics · Machine Reassignment · Virtual Machine Placement

1 Introduction

In service hosting and virtualized hosting, services or virtual machines must be assigned to clustered servers. Each server has to provide enough resources, such as cpu, ram or disk, in order to have all of its processes running. The machine reassignment problem, proposed by Google for the ROADEF/EURO challenge 2012, is such a problem, with some additional constraints and a cost function to minimize.

In this paper, we propose a modeling framework for these packing problems and a greedy heuristic framework to find feasible assignments. We adapt several classical bin packing heuristics and propose new variants. We provide a worst-case complexity analysis of these algorithms and present numerical results on different classes of randomly generated instances. We also highlight some structural properties of the machine reassignment problem and use them to adapt our heuristics. We provide experimental results on realistic instances for the machine reassignment problem.

1.1 Bin packing problems

In the classical Bin Packing (BP) problem, we are given a set $\mathcal{I} = \{I_1, \ldots, I_n\}$ of n items, a capacity $C \in \mathbb{N}$ and a size function $s : \mathcal{I} \to \mathbb{N}$. The goal is to find a feasible assignment minimizing the

*Laboratoire G-SCOP, UMR 5272, 46, avenue Félix Viallet - 38031 Grenoble Cedex 1 - France
michael.gabay@g-scop.grenoble-inp.fr

†UJF, Inria Grenoble, 655 avenue de l'Europe, Montbonnot 38334 Saint Ismier Cedex, France.
sofia.zaourar@inria.fr

number of bins used. A feasible assignment of the items into N bins is a partition $P_1, \ldots, P_N$ of the items, such that for each P_k, the sum of the sizes of the items in P_k does not exceed the capacity C. In the decision version of this problem, the number of bins N is part of the input and the objective is to decide if all of the items can be packed using at most N bins. This problem is known to be strongly NP-hard (Garey and Johnson, 1979).

Garey et al (1976) introduced a generalization of this problem, called Vector Bin Packing (VBP) or d-Dimensional Vector Packing (d-DVP). In this problem, item sizes are described by a d-dimensional vector: $(s_i^1, \ldots, s_i^d)$ and bins have a capacity C in all dimensions. A feasible assignment of the items into N bins is a partition $P_1, \ldots, P_N$ of the items such that for each P_k, on each dimension, the sum of the sizes of the items in P_k does not exceed the capacity:

$$\forall k \in \{1, \ldots, N\},\ \forall j \in \{1, \ldots, d\},\ \sum_{i \in P_k} s_i^j \leq C$$

Vector bin packing is often used to model virtual machine placements (Lee et al, 2011; Panigrahy et al, 2011; Stillwell et al, 2010). In such cases, all machines are supposed to have the same capacities. This could be the case when a new computer cluster is built. However, as it grows and servers are renewed, new machines are introduced and the cluster becomes heterogeneous.

Hence, we are interested in a further generalization of this problem, where each bin has its own vector of capacities $(c_k^1, \ldots, c_k^m)$ and the goal is to find a feasible packing of the items. We call this problem the Variable Size Vector Bin Packing (VSVBP) problem. This problem has not been studied yet to the best of our knowledge. VSVBP can be used to model previously mentioned virtual machine placement problems in a realistic heterogeneous environment.

1.2 Machine reassignment problem

The machine reassignment problem was proposed by Google for the 2012 ROADEF Challenge. This challenge was based on problems occurring in Google's data centers and realistic instances were provided. In the machine reassignment problem, a set of processes needs to be (re)assigned to a set of machines. There are m resources and each machine (resp. process) has its own capacity (resp. requirement) for each resource. There are also additional constraints presented in Section 4. The aim is to find a feasible assignment minimizing a weighted cost.

In the challenge, an initial feasible solution was provided. Therefore, local search based heuristics were a natural (and successful) approach. Local search aims at improving iteratively a given solution by applying small modifications, and is well-suited to quickly improve solutions of very large-scale problems. See Aarts and Lenstra (1997) survey on local search.

When using local search, the search space is limited by the initial solution and the set of accepted moves. This space can be enlarged by running several local searches, starting from diversified initial solutions. This is the idea behind the Greedy Randomized Adaptive Search Procedure (GRASP) (Feo and Resende, 1989, 1995). This approach is an iterative process where one successively creates a new feasible solution, then optimizes it using a local search heuristic. When applying a GRASP heuristic to the machine reassignment problem, VSVBP arises as a subproblem for generating new initial solutions. We explain how we can handle additional constraints and find new feasible assignments by solving VSVBP problems in Section 4.

1.3 Outline

In this paper, we study different heuristics to solve VSVBP and point out properties of the machine reassignment problem. In Section 2, we define VSVBP and present related works on vector bin packing. In Section 3, we propose several heuristics for this problem. In Section 4, we discuss structural properties of the machine reassignment problem and adapt our heuristics. Experimental results are reported Sections 3.5 and 4.5.

2 Variable size vector bin packing problem

An instance of the variable size vector bin packing problem is defined by C an $\mathbb{N}^{N\times d}$ capacity matrix and S an $\mathbb{N}^{n\times d}$ size matrix, where c_k^j (resp. s_i^j) denotes the capacity of bin k (resp. the size of item i) in dimension j.

We define the following index sets: $\mathcal{B} = \{1,\dots,N\}$ for the bins, $\mathcal{I} = \{1,\dots,n\}$ for the items, and $\mathcal{D} = \{1,\dots,d\}$ for the dimensions.

The problem is to find a feasible assignment $x \in \mathbb{N}^{n\times N}$ of the items into the bins such that:

$$\sum_{i\in\mathcal{I}} s_i^j x_{i,k} \;\leq\; c_k^j \qquad \forall j \in \mathcal{D},\ \forall k \in \mathcal{B} \tag{1}$$

$$\sum_{k\in\mathcal{B}} x_{i,k} \;=\; 1 \qquad \forall i \in \mathcal{I} \tag{2}$$

$$x_{i,k} \;\in\; \{0,1\} \qquad \forall i \in \mathcal{I},\ \forall k \in \mathcal{B} \tag{3}$$

Inequality (1) models the capacity constraints while constraints (2) and (3) ensure that each item is assigned to a bin.

2.1 Related work

Since VSVBP is a generalization of BP, this problem is strongly NP-hard. Moreover, Chekuri and Khanna (1999) proved that 2-DVP is APX-hard and showed $d^{1/2-\epsilon}$ hardness of approximation. Woeginger (1997) proved that there is no asymptotic PTAS (unless P=NP). Hence, as a generalization of d-DVP, the optimization version of VSVBP (with costs on the bins) is APX-hard and cannot have an asymptotic PTAS.

Maruyama et al (1977) generalized classical bin packing heuristics into a general framework for VBP.

There are many theoretical results for the vector bin packing problem: Kou and Markowsky (1977) studied lower and upper bounds and showed that the worst case performance ratio for the generalization of some classical bin packing algorithms is larger than d, where d is the dimension. Yao (1980) proved that any $o(n \log n)$ time algorithm has a worst case performance ratio bigger than d. Bansal et al (2006) proposed a randomized $(\log d + 1 + \epsilon)$-approximation. Their algorithm is polynomial for fixed d. Spieksma (1994) proposed two lower bounds for 2-DVP and a branch-and-bound algorithm using these bounds. Caprara and Toth (2001) analyzed several lower bound for 2-DVP and showed that the lower bound obtained by the linear programming relaxation of the (huge) integer programming formulation they propose, dominates all of these bounds. Chang et al (2005) used 2-DVP to model a packing problem where steel products have to be packed into special containers and they proposed a heuristic. Caprara et al (2003) showed that there is a PTAS for

d-DVP if all items sizes are totally ordered. Shachnai and Tamir (2003) studied Data Placement problem as an application of VBP and proposed a PTAS for a subcase of VBP. Karp et al (1984) studied VBP where all items sizes are drawn independently from the uniform distribution over [0,1]. They proved that the expected wasted space by the optimal solution is $\Theta(n^{\frac{d-1}{d}})$ and proposed an algorithm that tries to pack two items in each bin and has the same expected wasted space.

Stillwell et al (2010) implemented and compared several heuristics for VBP with additional real-world constraints in the case of virtualized hosting platforms. They found out that the algorithm which is performing the best is the choose pack heuristic from Leinberger et al (1999) with items sorted by decreasing order of the sum of their requirements.

Han et al (1994) studied 2-VSVBP optimization problem, with several types of available bins and the aim is to minimize the sum of bin costs. They proposed exact and heuristic approaches along with a process to improve lower bounds.

In the classical First Fit Decreasing (FFD) heuristic, one has to select the *largest* item and then pack it into a bin. Hence, if one generalizes this heuristic to the multidimensional case, it has to be determined how to measure and compare items. Panigrahy et al (2011) presented a generalization of the classical First Fit Decreasing (FFD) heuristic to VBP and experimented several measures. A promising measure is the $DotProduct$ which defines the *largest* item as the item that maximizes some weighted dot product between the vector of remaining capacities and the vector of requirements for the item.

3 Heuristic framework

We generalize the classical First Fit Decreasing (FFD) and Best Fit Decreasing (BFD) heuristics to VSVBP. Algorithm 1 is the classical BFD algorithm. Panigrahy et al (2011) proposed a different approach of this algorithm which focuses on the bins, as illustrated by Algorithm 2. In order to use these algorithms in multidimensional packing problems, one needs to define an ordering on bins and items.

This ordering can be defined using a measure: a size function which returns a scalar for each bin and item. In the following sections we propose several measures based on the remaining capacities of the bins and decisions made.

Since orderings are based on a measure, both the orderings of items and bins may change in the course of the algorithm. Remark that if the order is unchanged then item centric and bin centric heuristics give the same results (either both are infeasible or both are feasible and return the same solution).

Remark that any greedy algorithm for this problem can be reduced to a best fit item centric heuristic by computing the next decision of the algorithm in the measure and returning size 2 for chosen item, size 0 for chosen bin and size 1 for other bins and items.

3.1 Measures

In order to sort items and bins, we define a measure. Let $i \in \mathcal{I}$, $k \in \mathcal{B}$, $j \in \mathcal{D}$. We define $\mathcal{I}_r$ as the set of unpacked items and $\mathcal{B}_r$ as the set of remaining bins (unless we are using a bin centric approach, $\mathcal{B}_r = \mathcal{B}$). We denote by r_k^j the remaining capacity of bin k in dimension j, by $C(j)$ the total remaining capacity in dimension j and by $R(j)$ the total requirement in dimension j: $C(j) = \sum_{k \in \mathcal{B}_r} r_k^j$ and $R(j) = \sum_{i \in \mathcal{I}_r} s_i^j$.

Algorithm 1: BFD Item Centric

```
while There are unpacked items do
    Compute sizes
    Pack the biggest item into the smallest feasible bin
    if the item cannot be packed then
        return Failure
return Success
```

Algorithm 2: BFD Bin Centric

```
while The list of bins is not empty do
    Compute sizes
    Select b the smallest bin
    while An unpacked item fits into b do
        Compute sizes
        Pack the biggest feasible item into b
    Remove b from the list of bins
if An item has not been packed then
    return Failure
return Success
```

A natural idea to define a scalar size from vector size is to take a weighted sum of the vector components. We define the following sizes:

$$S_{\mathcal{B}}(k) = \sum_{j \in \mathcal{D}} \alpha_j s_k^j \qquad \forall k \in \mathcal{B}$$

$$S_{\mathcal{I}}(i) = \sum_{j \in \mathcal{D}} \beta_j r_i^j \qquad \forall i \in \mathcal{I}$$

where α and β are two scaling vectors. We propose three different scaling coefficients: $\frac{1}{C(j)}$, $\frac{1}{R(j)}$ and $\frac{R(j)}{C(j)}$. The first ratio normalizes based on bins capacities. The second ratio normalizes based on items requirements. The last coefficient takes both remaining capacities and requirements into account and normalizes on the rarity of resources.

We can also define the size of an item by choosing its maximal normalized requirement over the resources. We obtain the priority measure:

$$S_{\text{prio}}(i) = \max_{j \in \mathcal{D}} \frac{r_i^j}{C(j)} \qquad \forall i \in \mathcal{I}$$

Sizes are either computed once and for all, before the first run of the algorithm, in such case we say that the measure and the resulting heuristics are *static*, or at every iteration of the algorithm. In this latter case, we have *dynamic* measures and heuristics.

For static measures, the ordering is fixed and Algorithms 1 and 2 become first fit heuristics. Observe that for a same static measure, Algorithms 1 and 2 return the same results.

Both the *static* and *dynamic* heuristics are considered in this paper. In the following, we choose $\alpha = \beta$. As a consequence, the measure S has the following property:

Property 1. *If $\alpha = \beta$ and $S_{\mathcal{B}}(k) < S_{\mathcal{I}}(i)$ then the item i does not fit into the bin k.*

Proof. If $S_{\mathcal{B}}(k) < S_{\mathcal{I}}(i)$, then $\sum_{j \in \mathcal{D}} \alpha_r (r_k^j - s_i^j) < 0$. Since both r and s are positive, $r_k^j < s_i^j$ for some j. □

3.2 Bin balancing

In Section 3.1, we presented measures which yield different heuristics, using Algorithms 1 and 2. However, since bin capacities are different, it is hard to predict which resource, bin or item will be the bottleneck(s). Moreover, we can take advantage of the fact that we are only interested in finding feasible assignments. Instead of packing as many items as possible in a bin, we can try to balance the load. The Permutation Pack and Choose Pack heuristics from Leinberger et al (1999) use such an approach to pack items. We propose another approach: using the item centric heuristic, pack current item into the first feasible bin. Then, move this bin (or a subset of the bins) to the end of the list of bins. This approach is detailed in Algorithm 3. Line 7, l_B is updated by one of the two following ways:

- *Single bin balancing:* Used bin is moved to the end of the list
- *Bin balancing:* All bins tried (including the successful bin) are moved to the end of the list in the same order: let l be the new list. We have:
  $l(1) = l_B(j+1), \ldots,\ l(N-j) = l_B(N),\ l(N-j+1) = l_B(1), \ldots,\ l(N) = l_B(j)$
  (this is actually achieved through a simple modulo)

Algorithm 3: Bin Balancing Heuristics

```
1  Sort l_B (bins list) and l_I (items list)
2  while There are unpacked items do
3      Let I be the biggest unpacked item
4      for j = 1 to N do
5          if item I can be packed into l_B[j] then
6              Pack I into l_B[j]
7              Update l_B
8              break
9      if I has not been packed then
10         return Failure
11 return Success
```

The main idea of this algorithm is that once an item is assigned to a bin, we try to assign the following items to other bins, in order to prevent critical bins from being overwhelmed too early.

3.3 Dot Product

We generalize the *DotProduct* heuristic from Panigrahy et al (2011). In this heuristic we select the feasible pair (i, k) maximizing the (weighted) dot product $s_i \cdot r_k$ (resp. $\sum_{j \in \mathcal{D}} \alpha_{i,k} s_i^j r_k^j$) and pack item i into bin k.

We propose three variants of this heuristic: maximize the dot product ($\alpha_{i,k} = 1$), or the weighted dot product with $\alpha_{i,k} = (\|s_i\|_2 \|r_k\|_2)^{-1}$ or $\alpha_{i,k} = \|r_k\|_2^{-2}$.

On the first iteration, we compute dot products for all feasible pairs, then store these values. On the following iterations, only the dot products concerning the bin where an item has just been packed are computed. The worst case time and space complexity for initializing sizes is $\mathcal{O}(dnN \log(nN))$. The complexity of computing costs afterwards is at most $\mathcal{O}(dn)$ and the list can be maintained in $\mathcal{O}(n \log(nN))$.

This heuristic maximizes the *similarity* of a bin and an item (the scalar projection of the item sizes onto the bin remaining capacities). Moreover, we need to be able to compare these dot products for all pairs of bins and items. On one hand, if we do not scale the vectors, then we maximize both the similarity and the size used. On the other hand, if we normalize both sizes and capacities, we minimize the angle between the two vectors. Eventually, if we re-scale by $\frac{1}{\|r_k\|_2^2}$, then we focus on maximizing the scalar projection of the item and maximize similarity.

3.4 Complexity

We denote $p = \max(n, N)$. In the worst case scenario, both the item centric and the bin centric algorithms behave as shown in Algorithm 4. Hence, the overall time complexity is $\mathcal{O}(dp^2 + p^2 \log p)$. The space complexity is $\mathcal{O}(p^2 + dp)$ for the dot product and $\mathcal{O}(dp)$ for the other measures.

Algorithm 4: Worst-case heuristics behavior

```
1 Initialize sizes                          // O(dp²)  (DotProduct)
2 for i = 1 to p do                         // p×
3 |   Compute sizes                         //   O(dp)  (for given measures)
4 |   Sort lists                            //   O(p log p)
5 |   Pick an item                          //   O(1)
6 |   Pack it                               //   O(dp)
```

Obviously, one shall not implement the algorithm as described by Algorithm 4. When using a static measure, bins and items should only be sorted at the beginning of the algorithm. The overall complexity will be $\mathcal{O}(dp^2)$.

Moreover, when checking whether an item fits into a bin, we can stop on the first dimension where the remaining capacity is smaller than the size of the item. Furthermore, when using one of the measures described in Section 3.1 or any other measure verifying Property 1, we can use this property to avoid checking feasibility when $S_{\mathcal{I}}(i) > S_{\mathcal{B}}(k)$. These optimizations, however, do not improve the worst-case complexity of the algorithm.

3.5 Experiments

We experimented all described heuristics on 5 classes of generated instances. For each of these classes, we generated 100 feasible instances for each configuration with 10, 30 and 100 bins and 2, 5 and 10 dimensions. The whole test bench contains 4500 generated instances. In this section, we say that x% of bin k has been used if the average usage of the bin is more than x, *i.e.*

$\sum_{\text{s.t. } c_k^j \neq 0, j \in \mathcal{D}} (\frac{c_k^j - r_k^j}{c_k^j} - \frac{x}{100}) \geq 0.$

Instances. In the first class of instances (*Random uniform*), bin capacities are chosen independently using a uniform distribution on [10;1000]. Then, items sizes are independently drawn from a uniform distribution on $[0; 0.8 \times r_k^j]$ until at least 80% of the bin capacity is used.

The second class of instances (*Random uniform with rare resources*) is the same as the first, except that after generating the capacities of a bin, the capacity in dimension d is set to 0 with probability 0.25. Last dimension is a rare resource.

In the third class of instances (*Correlated capacities*), for each bin, an integer $b_0 \in [10; 1000]$ is uniformly generated. Then, each capacity $j \in \mathcal{D}$ is set to $0.9 \times b_0 + X_j$ where X_j is an exponentially distributed random variable with rate parameter $1/(0.1 \times b_0)$ (standard deviation is equal to 10% of b_0). Items sizes are generated as in the first and second classes.

Bins in the fourth class (*Correlated capacities and requirements*) are generated as in the third one. Items in this class are generated similarly to the bins, with $b_0 \in [1; 0.8 \times r_k^j]$ and until at least 80% of the bin capacity is used or we failed 100 times to generate a feasible item.

In the fifth class of instances (*Similar items and bins*), bin capacities are chosen uniformly and independently on [10;1000]. For each item, size in dimension j is set to $X_j + c_k^j/5$ where X_j is an exponentially distributed random variable with rate parameter $1/(0.2 \times c_k^j/5)$ (standard deviation is equal to 20% of $c_k^j/5$). Items are generated until at least 70% of the bin capacity is used or we failed 100 times to generate a feasible item.

In classes 1 to 4, on average, 85% of the bins capacities are used in generated instances. In class 5, 79% of bins capacities are used on average.

Heuristics. We use the following measures: *nothing* (static, items and bins are kept as provided in the input), static shuffle, static $1/C$, static $1/R$, static R/C, dynamic shuffle, dynamic $1/C$, dynamic $1/R$ and dynamic R/C. We use these measures with the item centric, bin centric, bin balancing and single bin balancing heuristics. This gives 31 heuristics since item centric and bin centric heuristics are the same for static measures. We also use the 3 variants of the dot product heuristic.

Heuristics and instance generator were implemented in Python and were not optimized. Still, except the dot product heuristics for which we unnecessarily recompute all sizes on all iterations, all of the heuristics run in less than 0.1 second on any of the instances (using PyPy interpreter).

In order to benchmark our heuristics, we compute the number of successes (number of feasible solutions) and the average percentage of items packed (we keep packing items even if we know that the solution will be infeasible). Figure 7 of Appendix A shows the total number of successes of all heuristics on the benchmark.

Results. We benchmarked our heuristics against random orderings and a first observation is that random heuristics achieve the worst performance. Among random heuristics, the static item centric ones are leading to better results than the other ones. This was expected since the static item centric heuristics corresponds to first fit heuristics while the other ones are just performing random assignments.

In Figures 1, 2, 3, 4 and 6, we show the heuristic achieving the highest total number of successes on this class for each family of heuristics.

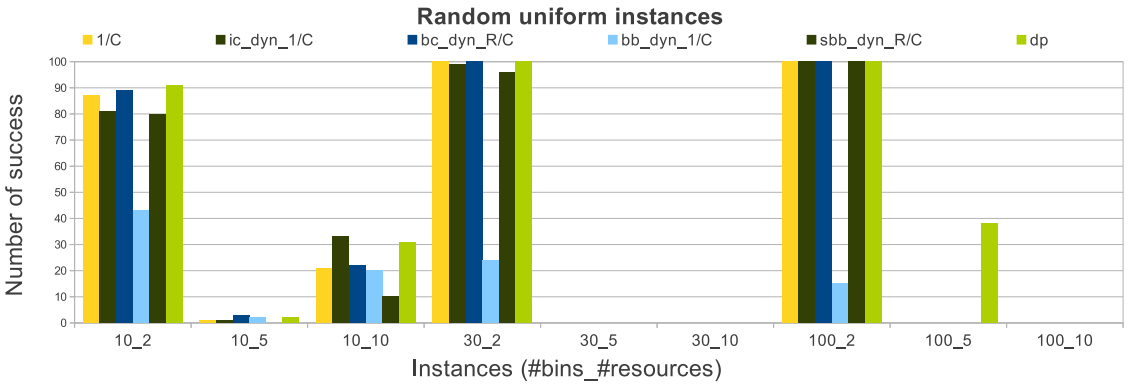

Figure 1: Random uniform instances, number of feasible solutions

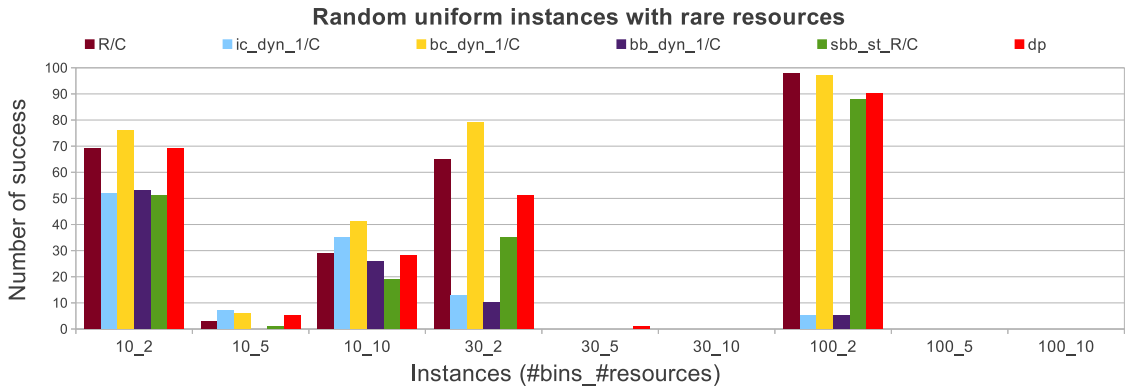

Figure 2: Random uniform instances with rare resources, number of feasible solutions

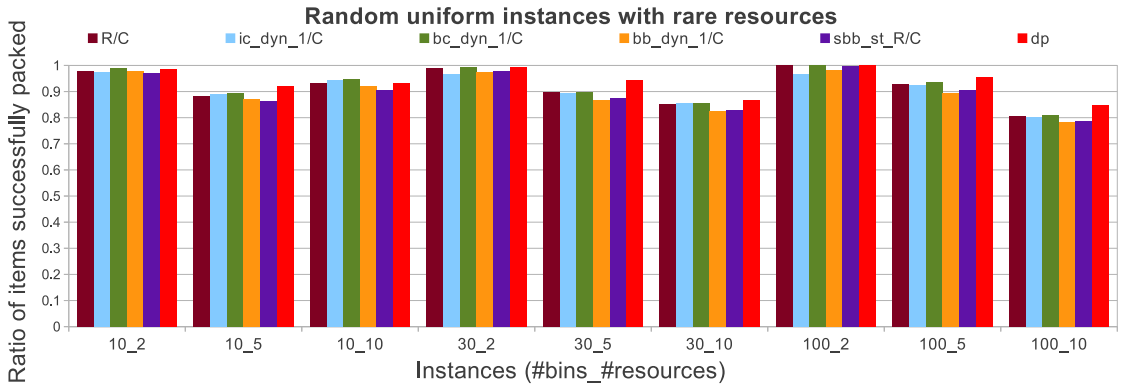

Figure 3: Random uniform instances with rare resources, average ratios of items packed

In Figure 1, on random uniform instances, we observe that static best fit, dynamic item centric, dynamic bin centric and dot product heuristics are roughly achieving the same performance. Yet, the dot product heuristic is the only heuristic providing feasible solutions with 100 bins and 5 resources. With a rare resource, in Figure 2, bin centric heuristics are providing slightly better results than other heuristics. Since bin centric heuristics focus on the bins rather than the items, they can make better use of the bins, especially if they first consider bins with null rare resources before considering bins providing rare resources. This helps to avoid getting stuck later on in the algorithm if items with null sizes in the rare resource were packed in bins providing this resource.

Paradoxically, with 10 bins, instances with 10 resources are easier to solve than instances with 5 resources. The reason is that the higher the number of resources, the lower is the probability that an item fits into a different bin other than its initial bin. With very few bins, this actually guides the heuristic.

In Figure 3, we observe that even though heuristics may not provide feasible solutions, 90% of the items are packed on average. The average percentage of items packed only depend on the class of instance, and not the heuristic used.

In Figure 4, we observe that when bin capacities are correlated, heuristics perform very well on instances with few resources while their performance drastically decrease as the number of resources increases. The dot product accounts for these correlations and achieves better performance than other heuristics.

When both capacities and item sizes are correlated, the problem is almost the same as the single dimensional bin packing decision problem. In Figure 5, we observe that all heuristics are performing very well except the random ones and the third dot product. In this latter case, remark that since items and bins are normalized, all items and bins are roughly the same to this heuristic, resulting in a random assignment.

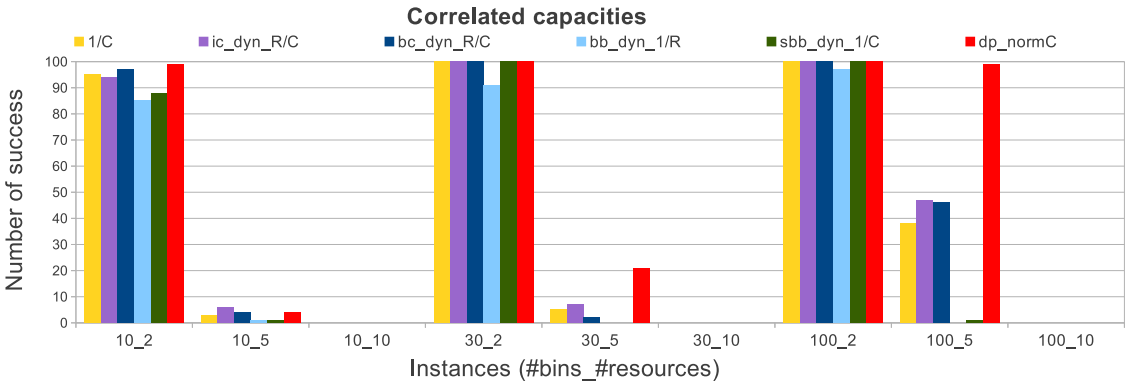

Figure 4: Correlated bin capacities, number of feasible solutions

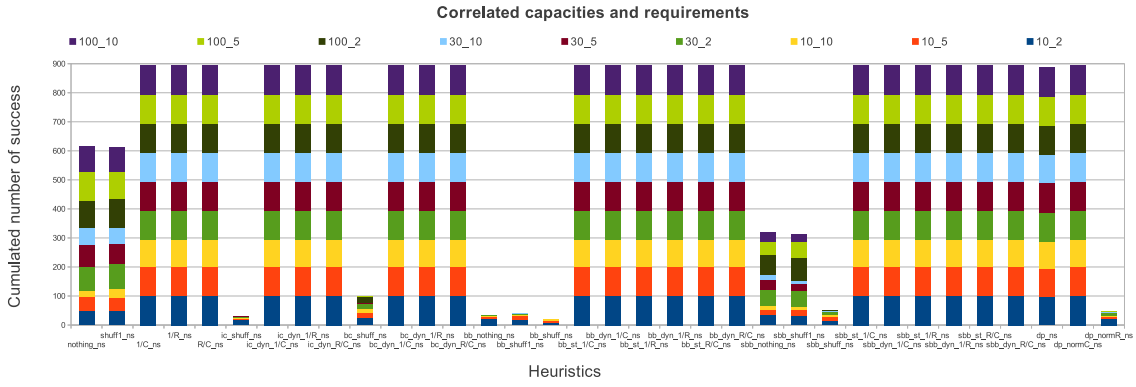

Figure 5: Correlated bin capacities and item sizes, cumulative results

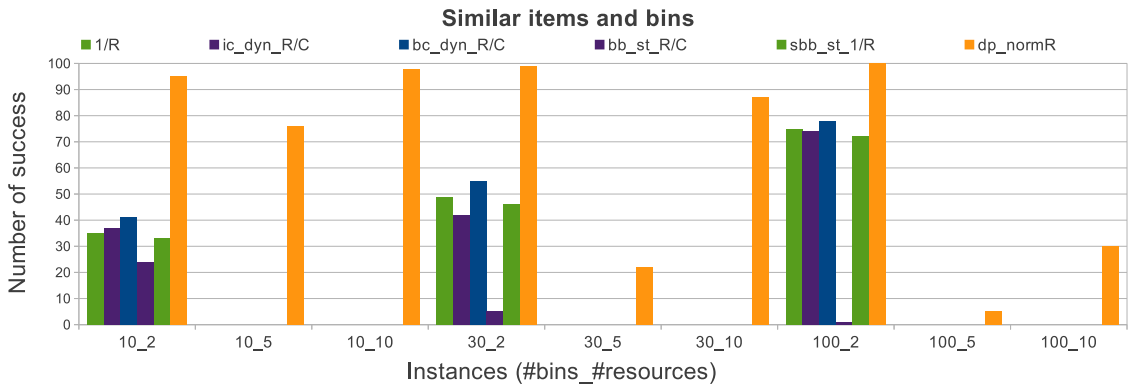

Figure 6: Similar bins and items, number of feasible solutions

For the similar instances, in Figure 6, the third dot product heuristic outperforms other heuristics (including the two other dot products). Remark that on the initial configuration (all items remaining and empty bins), the normalized dot product of an item with its initial bin will be close to 1 with high probability. Other heuristics are blind to this similarity criterion.

On this benchmark we observe that as the number of resources grows, the problem quickly becomes much harder. Moreover, dynamic item centric, dynamic bin centric and dot product heuristics outperform bin balancing heuristics in terms of number of feasible solutions found. If we analyze the data, we remark that if we combine the results of all heuristics from the same family, the total number of feasible solutions will almost remain the same. Yet bin balancing heuristics provide solutions on some instances which are infeasible for all other heuristics. Since all of these heuristics are very fast to compute, one can consider applying all of them to problem instances.

4 Application to the Machine Reassignment Problem

The machine reassignment problem is a simplified version of problems encountered with data centers: several processes are assigned to different servers, in several data centers, all over the world. The system needs to be robust to energy or machine failures. Moreover, some processes depend on each other and hence have to run on machines which are *close to* each other. Occasionally they consider moving processes to different servers in order to increase system performance. In the machine reassignment problem, system performance is modeled by an aggregated cost and the aim is to minimize it.

The variable size vector bin packing problem is a subproblem of the machine reassignment problem: any feasible assignment for the machine reassignment problem is a feasible VSVBP assignment for the problem defined with items sizes being processes requirements and bins capacities being the machines capacities. Yet, there are some additional constraints in the machine reassignment problem:

- *Conflict constraints*: Processes are partitioned into services and two processes of the same service cannot be assigned to the same machine.

- *Transient usage constraints*: when a process is moved from one machine to another, some resources (such as disk space) remain used on the first machine. Thus, the process consumes its requirement of these transient resources on both its initial and final machines.
- *Spread constraints*: Machines are partitioned into locations and each service s needs to have its processes spread over a minimum number of distinct locations, denoted $spreadMin(s)$.
- *Dependency constraints*: Machines are partitioned into neighborhoods and if a service s^a depends on a service s^b, then any process from s^a has to run on some machine having in its neighborhood a machine running a process from s^b.

The goal of the machine reassignment problem is to find a feasible assignment minimizing a weighted cost. The whole subject can be found on the challenge webpage[1].

In order to use a diversified multi-start approach, we need to get various, diversified, initial feasible solutions. We only consider feasibility and not solution costs. In this section, we highlight some structural properties of the machine reassignment problem and show how our heuristics can be adapted to this problem and its constraints.

We will use the subject notations in the remainder of this section: we denote by $\mathcal{M}$ the set of machines, $\mathcal{N}$ the set of neighborhoods, $\mathcal{P}$ the set of processes, $\mathcal{R}$ the set of resources, $\mathcal{TR} \subseteq \mathcal{R}$ the set of transient resources and $\mathcal{S}$ the set of services. $\mathcal{N}$ is a partition of $\mathcal{M}$ and $\mathcal{S}$ is a partition of $\mathcal{P}$. The function $N : \mathcal{M} \to \mathcal{N}$ maps each machine to its neighborhood. $M : \mathcal{P} \to \mathcal{M}$ is the assignment: it maps each process to its machine. M_0 denotes the initial assignment.
$R(p, r)$ is the requirement of resource $r \in \mathcal{R}$ for the process $p \in \mathcal{P}$. We denote by $C(m, r)$ the capacity of resource $r \in \mathcal{R}$ for the machine $m \in \mathcal{M}$. The two functions $R(r)$ and $C(r)$ are shorthands for $\sum_{p \in \mathcal{P}} R(p, r)$ and $\sum_{m \in \mathcal{M}} C(m, r)$, the overall requirement and capacity on resource r. We denote the initial amount of resource r consumed on machine m by

$$U_0(m, r) = \sum_{\substack{p \in \mathcal{P} \\ s.t.\ M_0(p) = m}} R(p, r)\,.$$

In this problem, we have an initial feasible solution which is used to define transient usage constraints. We will rely on this initial solution to derive properties and set a few process assignments.

In the following subsections, we present several properties of the machine reassignment problem and explain how we can use them to ensure that a feasible solution exists in the search space.

4.1 Transient usage constraints

Our heuristics can easily be adapted to integrate transient usage constraints. Indeed we can take them into account as follows: when initial bin capacities are set, let r_1 be a non-transient resource and r_2 a transient resource. For each machine $m \in \mathcal{M}$, we set its capacity in resource r_1 to $C(m, r_1)$ while we set its capacity in resource r_2 to $C(m, r_2) - U_0(m, r_2)$. Then, process requirements depend on machines: for all $r \in \mathcal{R} - \mathcal{TR}$ and all machines, they are equal to $R(p, r)$, while for all $r \in \mathcal{TR}$ they are equal to 0 for the machine $M_0(p)$ and to $R(p, r)$ otherwise. When a process is assigned to its initial machine, the capacity constraints on transient resources are always satisfied.

These constraints can be taken into account when sizes are computed. We can decide, for instance, that processes with huge requirements on some transient resources will not be moved.

[1] `http://challenge.roadef.org/2012/en/`

Moreover, remark that if a process is moved from its initial machine, then for all of its transient resources, the space used is lost. Hence, we have the following property:

Property 2. *For each process $p \in \mathcal{P}$, if there is a transient resource $r \in \mathcal{TR}$ such that $R(p,r) > C(r) - R(r)$, then in every feasible assignment, p has to be assigned to its initial machine.*

Proof. Let p be a process and r a transient resource such that $R(r) > C(r) - R(p,r)$. If process p is moved, since r is transient, a space $R(p,r)$ on machine m cannot be used by any process. Hence, the total available space for all processes in resource r is $C(r) - R(p,r)$, which is smaller than the total requirement. Therefore, any assignment M with $M(p) \neq M_0(p)$ is not feasible. □

Using Property 2, we can determine that some processes cannot be moved. In such cases, we can fix them to their initial machines. If we are interested in moving a set of processes P, then we obtain the following corollary:

Corollary 1. *Let $P \subseteq \mathcal{P}$ be a subset of processes. If there is a transient resource $r \in \mathcal{TR}$ such that $\sum_{p \in P} R(p,r) > C(r) - R(r)$, then in every feasible assignment, at least one process from P is assigned to its initial machine.*

In a greedy approach, Property 2 and Corollary 1 can be used with C and R, the *residual* capacities and requirements. Moreover, they allow us to fix items and to conclude – before being unable to pack an item – that an intermediate solution (a partial assignment) is infeasible. Note that Property 2 and its corollary can be used during the optimization phase as well.

4.2 Conflict constraints

In order to satisfy conflict constraints, when trying to assign a process p from a service s to a machine m, one just needs to check that there is no process from service s which is already assigned to m.

Remark VSVBP can be reduced to VBP with conflict constraints. Indeed, let $c_{\max} = \max_{k \in \mathcal{B}, j \in \mathcal{D}} c_k^j$. Set bins capacities in VBP to $c_{\max}$. Add N conflicting items $p_{n+1}, \ldots, p_{n+N}$, with requirements $s_{n+k}^j = c_{\max} - c_k^j$. Any feasible solution to this VBP problem with conflict constraints gives a feasible assignment for the VSVBP problem.

4.3 Spread constraints

A simple way to make sure that these constraints are satisfied is the following: for each service $s \in \mathcal{S}$, take a subset of processes $P \subseteq s$ such that $|P| = spreadMin(s)$, and assign all processes of P to distinct locations. To make sure that there is a feasible solution, we use the initial solution to choose a subset of processes which will be assigned to their initial machines.

4.4 Dependency constraints

Dependency constraints are difficult constraints to cope with, because they bound processes to each other and can be cyclic. We propose to take advantage of these constraints to decompose the problem into smaller subproblems where all dependency constraints are satisfied. More precisely, let $g \in \mathcal{N}$ be a neighborhood, $m_1, m_2 \in g$ and $p \in \mathcal{P}$. Remark that if M is a feasible assignment with $M(p) = m_1$, then, setting $M(p) = m_2$ does not violate any dependency constraint. We can even generalize this property to all the processes from any neighborhood into any other neighborhood:

Property 3. *Let M be a feasible assignment. Denote by P_n the set of processes assigned to neighborhood $n \in \mathcal{N}$: $P_n = \{p \in \mathcal{P} : M(p) \in n\}$. Any assignment M' such that $\forall n \in \mathcal{N},\ \forall p_1, p_2 \in P_n,\ N(M'(p_1)) = N(M'(p_2))$, satisfies all dependency constraints.*

Proof. Let $s^a, s^b \in \mathcal{S}$, s^a depends on s^b. Let $p \in s^a$. The assignment M is feasible, hence $\exists p' \in s^b$ such that $M(p') \in N(M(p))$. Moreover $p, p' \in P_{N(M(p))}$. Therefore $p' \in N(M'(p))$. □

Property 3 implies that if one takes all processes from a given neighborhood and reassign all of them to a same neighborhood, then the new assignment satisfies all dependency constraints.

We use Property 3 with $M = M_0$ to decompose the problem into several subproblems where we either try to find an assignment for all processes from a given neighborhood into itself, or into another. In this latter case, recall that all transient resources used by the processes are lost. Hence, we have to make sure that Corollary 1 does not immediately induce that there is no feasible assignment. Moreover, such reassignment also implies that every process will be moved, possibly resulting in huge move costs.

4.5 Experiments

In this section, we apply several variants of VSVBP heuristics to machine reassignment problems.

Test problems. We use the 30 instances (sets A, B and X) provided during the ROADEF/EURO challenge. They are realistic instances, randomly generated according to real-life Google statistics. The largest instances contain up to $5,000$ machines, $50,000$ processes and 12 resources. More details on the instances can be found on the challenge webpage[2].

Implemented heuristics. Combining the above ideas to handle the additional constraints, our algorithm proceeds as follows. First, some processes are assigned to their initial machines in order to satisfy the spread constraints. In our experiments, on average 26% of the processes are assigned during this phase. Then, we decompose the problem into smaller independent subproblems. We define a subproblem by selecting all processes initially assigned to a neighborhood and the aim is to find a feasible assignment of these processes into this neighborhood. This makes dependency constraints automatically satisfied by any feasible assignment of the subproblems. We apply our various VSVBP heuristics to each neighborhood. Conflict and transient usage constraints are checked on the fly. Finally, the subproblems assignments are combined to form the global assignment.

We implemented the different types of VSVBP heuristics: item centric, bin centric and bin balancing. For each type, we used several measures, including the static $1/C$, $1/R$ and R/C measures, and the dynamic dot product and process priority measures. We also combined these measures with random orderings. In this case, we report the average results over 50 runs.

We implemented these heuristics in C++ using efficient data structures. Although there is still room for code optimization, we will see below that most heuristics are already very fast.

Results. In order to compare the different heuristics even on instances where they do not find feasible assignments, we report the percentage of assigned processes. 100% means that we found a feasible solution. Table 1 presents for each heuristic type, the best results obtained among the different measures, for each problem instance. The assignment column values are in bold font for the highest assignment percentage over the heuristics. The results of other implemented variants are reported in Tables 2 and 3 of Appendix B.

[2] `http://challenge.roadef.org/2012/files/Roadef%20-%20results.pdf`

In Table 1, we see that best performing heuristics are the bin balancing heuristics, with priority measure for processes and random machine order, or $1/C$ measure for machines and random processes order. These variants successfully assign more than 98% of the processes in average. Since the priority measure is updated after each assignment, this variant is way slower than the other one. The bin centric heuristic with machines and processes ordered randomly gives an average of 97.7% of processes assigned. Item centric heuristics with priority measure for the processes and either $1/C$ measure or random order for the machines give a similar average result. However, again, the running time is higher for this latter because of the priority measure cost. Note that for instance a_1_4, since each neighborhood is reduced to one machine, our neighborhood decomposition makes all the heuristics find the initial solution.

Now combining the different heuristics, we see that a feasible assignment is found for 16/30 instances, with 2/10 for A instances, and 8/10 for both B and X instances. We observe that our heuristics are likely to find solutions when R(r)/C(r) is below 85% on average.

Remark that even in the worst case (instance a_1_2), 92.9% of the processes are assigned. In such situations, if we optimize machine utilization, we may be able to assign the remaining processes.

	Bin balancing				**Bin centric**		**Item centric**			
	Prio Proc Rand Mach		Rand Proc $1/C$ Mach		Rand Proc Rand Mach		Prio Proc Rand Mach		Prio Proc $1/C$ Mach	
Pbs	%	time	%	time	%	time	%	time	%	time
a_1_1	**99**	0.00	**99**	0.00	**99**	0.00	97	0.00	98	0.00
a_1_2	90.2	0.02	89.8	0.00	91.9	0.00	92.1	0.02	**92.9**	0.02
a_1_3	96.1	0.00	95.9	0.00	96.7	0.00	**97.5**	0.00	97.2	0.00
a_1_4	**100**	0.00	**100**	0.00	**100**	0.00	**100**	0.00	**100**	0.00
a_1_5	96.3	0.01	96.4	0.00	96.7	0.00	**99.9**	0.01	97.2	0.01
a_2_1	98.7	0.02	98.7	0.00	98.2	0.00	96.4	0.02	96.9	0.02
a_2_2	96.1	0.01	96	0.00	96.7	0.00	96.4	0.01	96.8	0.01
a_2_3	96.5	0.01	96.5	0.00	97.1	0.00	96.9	0.01	97.1	0.01
a_2_4	96.5	0.01	96.2	0.00	96.4	0.00	**97.5**	0.01	96.7	0.01
a_2_5	95.3	0.01	95.3	0.00	95.1	0.00	94.2	0.01	95.3	0.01
b_1	97.3	0.25	96.8	0.00	97.2	0.01	97.1	0.25	97.5	0.26
b_2	92	0.24	**93.1**	0.00	85.8	0.01	87.8	0.24	86	0.24
b_3	99.8	3.44	99.8	0.01	99.9	0.02	99.9	3.26	99.9	3.43
b_4	**100**	1.32	**100**	0.01	99.9	0.06	**100**	1.37	99.9	1.39
b_5	99.9	16.06	99.9	0.03	**100**	0.05	**100**	14.64	**100**	16.00
b_6	**100**	8.71	**100**	0.02	**100**	0.07	**100**	8.54	**100**	8.77
b_7	99.7	8.65	99.7	0.03	**100**	0.92	**100**	9.17	**100**	9.41
b_8	99.9	17.94	**100**	0.03	99.9	0.05	99.8	15.38	**100**	17.73
b_9	**99.9**	6.45	99.9	0.02	98.6	0.31	97.8	6.63	97.7	6.87
b_10	99.9	6.41	99.9	0.03	**100**	0.92	**100**	7.33	**100**	7.54
x_1	96.6	0.25	96.6	0.00	96.8	0.01	**97.2**	0.25	97.1	0.25
x_2	92.8	0.24	92.9	0.00	85.9	0.01	86.1	0.24	85.7	0.24
x_3	99.8	3.43	99.8	0.01	99.9	0.02	99.9	3.24	99.9	3.43
x_4	**100**	1.10	**100**	0.01	**100**	0.06	**100**	1.15	**100**	1.18
x_5	99.9	15.98	99.9	0.03	**100**	0.05	**100**	14.53	**100**	15.91
x_6	**100**	8.43	**100**	0.02	**100**	0.07	**100**	8.27	**100**	8.48
x_7	99.6	9.08	99.6	0.03	**100**	0.95	**100**	9.63	**100**	9.87
x_8	99.9	18.08	**100**	0.03	**100**	0.05	**100**	15.44	**100**	17.83
x_9	**99.9**	6.29	**99.9**	0.02	99	0.31	98.2	6.46	98.6	6.68
x_10	99.9	6.25	99.9	0.03	**100**	0.92	**100**	7.12	**100**	7.33
avg/sum	98.1	138.6	98.1	0.38	97.7	4.8	97.7	133.2	97.7	142.9

Table 1: Results of the three types of proposed VSVBP heuristics on ROADEF/EURO challenge machine reassignment instances. For each heuristic type, the best variants results are reported, in terms of percentage of processes assigned (column "%") and CPU time (in seconds)

5 Conclusion

Our paper introduces the VSVBP problem, a generalization of the VBP which allows to account for many real-life problems. We propose a family of heuristics for the VSVBP, including adaptation of the well-known first fit and best fit bin packing heuristics, and some new variants taking advantage of the multidimensional resources and variable bin sizes. These heuristics are flexible and easy to implement.

We analyze the machine reassignment problem and highlight some of its properties. We use these properties to adapt our heuristics to this problem. We generate diversified feasible solutions that can be used as starting points for local search heuristics for instance.

Our code is open source and publicly available[3].

In future works, one can experiment more sophisticated measures, possibly based on the LP relaxation or the *permutation pack* and *choose pack* heuristics from Leinberger et al (1999). We can also reason on partial solutions and infer that some items have to be packed in a subset of the remaining bins. We can use constraint programming to implement this approach: propagate decisions taken by the heuristic, then take the next decision using updated domains.

References

Aarts E, Lenstra JK (1997) Local search in combinatorial optimization. John Wiley & Sons, Inc.

Bansal N, Caprara A, Sviridenko M (2006) Improved approximation algorithms for multidimensional bin packing problems. In: Foundations of Computer Science, 2006. FOCS'06. 47th Annual IEEE Symposium on, IEEE, pp 697–708

Caprara A, Toth P (2001) Lower bounds and algorithms for the 2-dimensional vector packing problem. Discrete Applied Mathematics 111(3):231–262

Caprara A, Kellerer H, Pferschy U (2003) Approximation schemes for ordered vector packing problems. Naval Research Logistics (NRL) 50(1):58–69

Chang SY, Hwang HC, Park S (2005) A two-dimensional vector packing model for the efficient use of coil cassettes. Computers & operations research 32(8):2051–2058

Chekuri C, Khanna S (1999) On multi-dimensional packing problems. In: Proceedings of the tenth annual ACM-SIAM symposium on Discrete algorithms, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, SODA '99, pp 185–194

Feo TA, Resende MG (1989) A probabilistic heuristic for a computationally difficult set covering problem. Operations research letters 8(2):67–71

Feo TA, Resende MG (1995) Greedy randomized adaptive search procedures. Journal of global optimization 6(2):109–133

Garey MR, Johnson DS (1979) Computers and intractability, vol 174. Freeman New York

Garey MR, Graham RL, Johnson DS, Yao AC (1976) Resource constrained scheduling as generalized bin packing. Journal of Combinatorial Theory 21:257–298

[3] `https://github.com/TeamJ19ROADEF2012/ROADEF2012-J19`

Han BT, Diehr G, Cook JS (1994) Multiple-type, two-dimensional bin packing problems: Applications and algorithms. Annals of Operations Research 50(1):239–261

Karp RM, Luby M, Marchetti-Spaccamela A (1984) A probabilistic analysis of multidimensional bin packing problems. In: Proceedings of the sixteenth annual ACM symposium on Theory of computing, ACM, pp 289–298

Kou LT, Markowsky G (1977) Multidimensional bin packing algorithms. IBM Journal of Research and development 21(5):443–448

Lee S, Panigrahy R, Prabhakaran V, Ramasubramanian V, Talwar K, Uyeda L, Wieder U (2011) Validating heuristics for virtual machines consolidation. Microsoft Research, MSR-TR-2011-9

Leinberger W, Karypis G, Kumar V (1999) Multi-capacity bin packing algorithms with applications to job scheduling under multiple constraints. In: Parallel Processing, 1999. Proceedings. 1999 International Conference on, IEEE, pp 404–412

Maruyama K, Chang S, Tang D (1977) A general packing algorithm for multidimensional resource requirements. International Journal of Computer & Information Sciences 6(2):131–149

Panigrahy R, Talwar K, Uyeda L, Wieder U (2011) Heuristics for vector bin packing. Tech. rep., Microsoft Research

Shachnai H, Tamir T (2003) Approximation schemes for generalized 2-dimensional vector packing with application to data placement. In: Approximation, Randomization, and Combinatorial Optimization.. Algorithms and Techniques, Springer, pp 165–177

Spieksma FC (1994) A branch-and-bound algorithm for the two-dimensional vector packing problem. Computers & operations research 21(1):19–25

Stillwell M, Schanzenbach D, Vivien F, Casanova H (2010) Resource allocation algorithms for virtualized service hosting platforms. Journal of Parallel and Distributed Computing 70(9):962–974

Woeginger GJ (1997) There is no asymptotic ptas for two-dimensional vector packing. Information Processing Letters 64(6):293–297

Yao ACC (1980) New algorithms for bin packing. Journal of the ACM (JACM) 27(2):207–227

A Comparison of all heuristics on random VSVBP instances

Figure 7 shows the total number of successes on all instances. We observe that random heuristics are outperformed by all other heuristics. Even if the dot product heuristic has the highest total number of successes, it does not significantly outperform other heuristics.

B Heuristics detailed results on machine reassignment problems

In this section, we give the results for some variants of VSVBP heuristics, adapted to the machine reassignment problem. Table 2 reports the performance of several bin balancing heuristics with different measures, in terms of number of assigned processes and running time. Similarly, Table 3 presents the performances of several bin centric heuristic variants.

The two last variants in Table 2 as well as the fourth variant in Table 3 give results close to the best heurictics presented Table 1. More generally, observe that all variants find feasible assignment for some instances.

Regarding CPU time, bin balancing variants with static measures are the fastest: less than one second to solve all the instances. Bin centric static variants take a few seconds. As expected, the slowest variants are the ones using dynamic measures. In particular, bin centric dot product heuristic does not scale well to large instances.

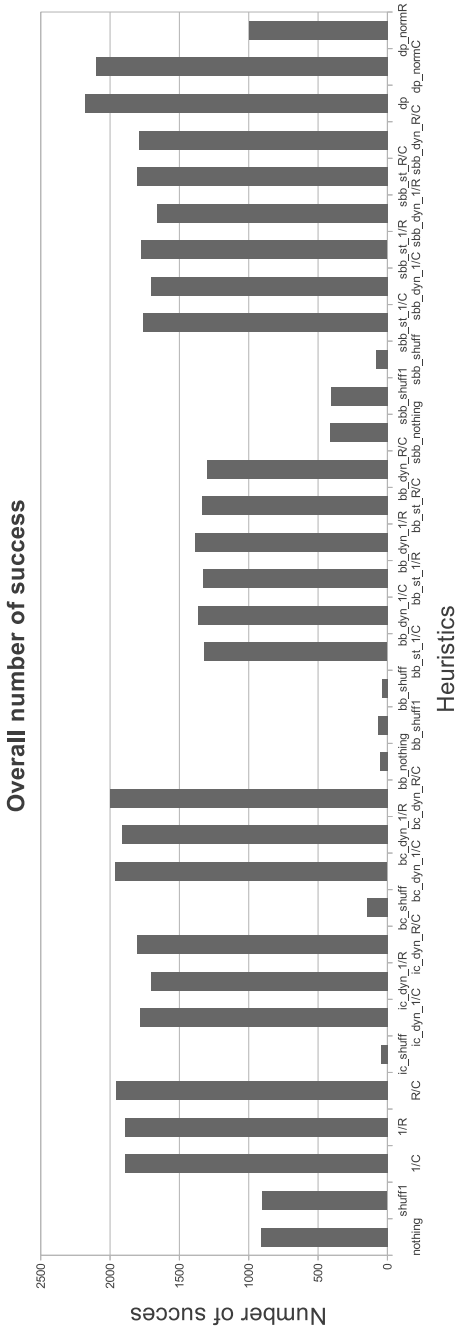

Figure 7: Total number of successes of the 34 heuristics, on the 4500 generated input

	Bin balancing variants									
	$1/C$ Proc $1/C$ Mach		$1/R$ Proc $1/C$ Mach		$1/C$ Proc Rand Mach		Rand Proc Rand Mach		Prio Proc $1/C$ Mach	
Pbs	%	time	%	time	%	time	%	time	%	time
a_1_1	97	0.00	95	0.00	91	0.00	**99**	0.00	**99**	0.00
a_1_2	76.6	0.00	77.3	0.00	77.5	0.00	89.8	0.00	88.6	0.02
a_1_3	94.5	0.00	93.7	0.00	95	0.00	95.8	0.00	96.2	0.00
a_1_4	**100**	0.00	**100**	0.00	**100**	0.00	**100**	0.00	**100**	0.00
a_1_5	81.8	0.00	85.4	0.00	77.7	0.00	96.5	0.00	95.7	0.01
a_2_1	62.7	0.00	61.5	0.00	61.5	0.00	98.6	0.00	98.6	0.02
a_2_2	96	0.00	97.3	0.00	97.1	0.00	96	0.00	95.6	0.01
a_2_3	97.2	0.00	97.2	0.00	97.4	0.00	96.6	0.00	95.9	0.01
a_2_4	93.8	0.00	88.1	0.00	90.9	0.00	96.2	0.00	96.1	0.01
a_2_5	84.7	0.00	85.6	0.00	87.2	0.00	95.2	0.00	**95.9**	0.01
b_1	82.8	0.00	82.5	0.00	83.4	0.00	96.8	0.00	**97.6**	0.25
b_2	58.5	0.00	59.8	0.00	60.9	0.00	92.9	0.00	90	0.24
b_3	99.8	0.02	**100**	0.02	99.5	0.02	99.8	0.01	99.9	3.37
b_4	92.6	0.02	94.7	0.02	94	0.02	**100**	0.01	**100**	1.31
b_5	**100**	0.03	**100**	0.03	**100**	0.03	99.9	0.03	99.9	15.05
b_6	**100**	0.02	**100**	0.02	**100**	0.02	**100**	0.02	**100**	8.66
b_7	**100**	0.03	99.9	0.03	99.9	0.03	99.7	0.03	99.7	8.57
b_8	**100**	0.04	**100**	0.04	**100**	0.04	**100**	0.03	**100**	16.02
b_9	72.6	0.27	72.4	0.27	73.9	0.27	99.9	0.03	99.9	6.42
b_10	**100**	0.02	**100**	0.03	**100**	0.03	99.9	0.03	99.9	6.38
x_1	82.1	0.00	84.2	0.00	82	0.00	96.6	0.00	95.8	0.25
x_2	62.8	0.00	60.8	0.00	59.8	0.00	92.4	0.00	**94.3**	0.24
x_3	99.7	0.02	**100**	0.02	99.4	0.02	99.8	0.01	99.8	3.34
x_4	95.2	0.02	92.3	0.02	95.9	0.02	**100**	0.01	**100**	1.10
x_5	**100**	0.03	**100**	0.03	**100**	0.03	99.9	0.03	99.9	14.95
x_6	**100**	0.02	**100**	0.02	**100**	0.02	**100**	0.02	**100**	8.39
x_7	99.7	0.04	99.8	0.04	99.6	0.04	99.6	0.04	99.6	9.03
x_8	**100**	0.04	**100**	0.04	**100**	0.04	**100**	0.03	99.9	16.06
x_9	73.1	0.25	74.7	0.24	77.6	0.23	99.9	0.03	99.9	6.26
x_10	**100**	0.02	**100**	0.02	**100**	0.03	99.9	0.03	99.9	6.22
avg/sum	90.1	0.92	90.1	0.92	90	0.93	98	0.41	97.9	132.19

Table 2: Results of bin balancing heuristics using different measures, on ROADEF/EURO challenge machine reassignment instances. For each variant, the percentage of processes successfully assigned (column "%") and the CPU time (in seconds) are reported.

	Bin centric variants									
	$1/C$ Proc $1/C$ Mach		$1/R$ Proc $1/C$ Mach		$1/C$ Proc Rand Mach		Rand Proc $1/C$ Mach		Dot prod Proc $1/C$ Mach	
Pbs	%	time	%	time	%	time	%	time	%	time
a_1_1	88	0.00	88	0.00	89	0.00	98	0.00	88	0.00
a_1_2	79.3	0.00	79.3	0.00	79.8	0.00	92.2	0.00	80.1	0.13
a_1_3	94	0.00	94.7	0.00	96	0.00	96.5	0.00	95.2	0.01
a_1_4	**100**	0.00	**100**	0.00	**100**	0.00	**100**	0.00	**100**	0.00
a_1_5	74.4	0.00	78	0.00	76	0.00	94.8	0.00	81.7	0.02
a_2_1	**100**	0.00	**100**	0.00	**100**	0.00	97.9	0.00	**100**	0.73
a_2_2	**98.3**	0.00	**98.3**	0.00	98	0.00	96.8	0.00	97.8	0.01
a_2_3	98.6	0.00	98.7	0.00	98.7	0.00	96.7	0.00	**99.1**	0.01
a_2_4	95	0.00	95.3	0.00	92.8	0.00	96.2	0.00	94.4	0.01
a_2_5	89.1	0.00	88.3	0.00	87.7	0.00	95.2	0.00	89.4	0.01
b_1	81.5	0.01	81.5	0.01	82.3	0.01	96.7	0.00	74.7	0.69
b_2	57.6	0.01	57.1	0.01	57.9	0.01	86.6	0.01	57.5	0.80
b_3	99.4	0.02	99.4	0.03	99.7	0.02	99.9	0.02	96.2	13.67
b_4	96.8	0.09	97.4	0.09	96.6	0.09	**100**	0.06	89.3	36.21
b_5	**100**	0.05	**100**	0.05	**100**	0.05	**100**	0.05	98.4	65.07
b_6	73.3	0.11	71.1	0.12	71.6	0.12	**100**	0.07	72.3	128.21
b_7	**100**	1.47	**100**	1.45	**100**	1.28	**100**	0.90	**100**	1886.06
b_8	**100**	0.07	**100**	0.07	**100**	0.06	99.9	0.05	99.9	61.17
b_9	80	0.50	79.5	0.51	83.1	0.50	98.5	0.31	86.8	397.66
b_10	**100**	1.56	**100**	1.55	**100**	1.32	**100**	0.91	**100**	1368.69
x_1	80	0.01	80.8	0.01	81.1	0.01	96.7	0.00	79.5	0.63
x_2	58.6	0.01	58.4	0.01	58	0.01	86	0.01	56.5	0.84
x_3	99.1	0.02	98.8	0.03	99.5	0.02	99.9	0.02	90.7	14.79
x_4	96.2	0.09	95.2	0.09	98.9	0.08	**100**	0.05	89.2	29.94
x_5	**100**	0.05	**100**	0.05	**100**	0.05	**100**	0.05	96.3	66.74
x_6	70.6	0.11	69.1	0.12	70.8	0.12	**100**	0.07	71.9	124.91
x_7	**100**	1.52	**100**	1.52	**100**	1.33	**100**	0.95	**100**	2012.85
x_8	**100**	0.06	**100**	0.06	**100**	0.06	**100**	0.05	98.1	58.98
x_9	89.6	0.51	89.6	0.51	91.7	0.49	99.1	0.31	91.2	381.08
x_10	**100**	1.53	**100**	1.50	**100**	1.31	**100**	0.90	**100**	1310.05
avg/sum	90	7.81	90	7.77	90.3	6.95	97.6	4.81	89.1	7960.00

Table 3: Results of bin centric heuristics using different measures, on ROADEF/EURO challenge machine reassignment instances. For each variant, the percentage of processes successfully assigned (column "%") and the CPU time (in seconds) are reported.