A Genetic Algorithm to Solve the Virtual Machines Resources Allocation Problem in Multi-tier Distributed Systems

Paolo Campegiani
Università di Roma Tor Vergata
Dipartimento di Informatica, Sistemi e Produzione
Via del Politecnico 1, 00133 Roma
campegiani@ing.uniroma2.it

Abstract

We propose a genetic algorithm to find the optimal allocation of virtual machines in a multi-tier distributed environment. We consider a formal model which allows both for quantitative and qualitative resources, handles multiple SLAs and easily capture the structure of a distributed infrastructure. We show that this allocation problem is NP-hard, so we focus on approximate solutions including a genetic algorithm that appears to perform well in finding an approximate solution.

1. Introduction

As the support for virtual machines execution is finding its way in mainline, off-the-shelf hardware components, the focus for vendors and systems architects is progressively shifting towards the definition of management tools to enable the adoption of virtualization not only for consolidation of legacy system but as an architectural asset on which to build a distributed system upon. The promise of virtualization is that, by the definition of an intermediate layer between the operating systems (that is in direct control of the hardware) and the applications, we have a central management function that makes easier to implement desired features for a modern distributed system as load balancing, high availability, rapid infrastructure deployment. Virtualization doesn't come for free, and an efficient implementation of it all over a distributed architecture requires to extend many existing tools and methodologies to cope with new and increased management complexities.

In this paper, we consider a general model for virtual machines resources allocation where each virtual machine and each physical host is described by a multi-dimensional resource vector. Resources are both quantitative and qualitative, also allowing for different service level agreements (SLAs) to be considered. When we instantiate a machine (and a specific SLA) over a physical host we gain some profit, and we want to maximize the total profit while minimizing the number of required physical hosts. In this article we discuss a genetic algorithm to find an approximate solution for the model we presented in [4].

A genetic algorithm searches solutions incrementally: at the beginning of the algorithm the population elements (i.e. the set of solutions under evaluation) change very fast, then the rate of changes slows as we progress toward the optimal. This behaviour could be particulary beneficial in a virtualized architecture: differences in the allocation of virtual machines that stems from differences in the solution of the problem could be smoothly performed by using live migration and dynamic resources tuning. This incremental approach, plus the relative simplicity by which a genetic algorithm can be parallelized, makes this methodology promising for virtualized architectures.

In a multi-tier distributed systems, nodes belonging to the same tier are all functionally equivalent, and have different resources characterization: a front-end tier faces connections from clients and requires more network bandwidth than a storage tier which requires a predominant I/O path to the storage subsystem. To tackle all these different characterizations, a resources allocation model must be multi-dimensional, and this feature is one of the key element of our approach that differentiates it from previously published works. Our model also allows for qualitative resources modeling. As an example, we can spread the virtual machines between two geographical sites, by defining a qualitative resource like "being in the first/second area" only at a small marginal cost of more computational complexity.

The rest of the paper is organized as follows. In section 2 we briefly discuss related works. In section 3 we define the model both informally and as a linear programming problem, showing that it's a generalization of the classical knapsack problem [9]. In section 4 we expose our genetic algorithm, and results are presented in section 5. We conclude on section 6.

2. Related Works

Resources allocation of virtual machines is the object of many studies.

In [11] it's exposed a control of CPU shares of two competing virtual machines over the same physical node, with a control-theory based approach.

The model presented in [8] is 2-dimensional, as it consider a load dependent resource as the CPU and a load independent resource as the main memory to characterize the virtual machines, while other studies usually consider the CPU only, severely limiting their applicability in a real environment.

In [10] the CPU shares are dynamically allocated with the goal to optimize a global utility function, under varying workload levels, and in [3] the proposed architecture involves different Application Environments (AEs), each one comprising several physical machines bounded together. Each AE serves different classes of transactions, and server could be moved from one AE to another, to optimize a global utility function which is based on the performance metrics of the AEs, like response time and throughput. The proposed solver searches for the optimal number of physical servers for each AE, with a beam search algorithm.

In [13] a similar architecture has been considered for appliance-based autonomic provisioning. The architecture defines some Virtual Application Environments (VAEs): a VAE spans over one or more virtual servers, and each virtual server is defined inside a physical machine. Each VAE has a On-Demand Router that dispatches incoming requests to the less loaded virtual server inside the VAE, in a round-robin fashion. A global and utility-driven model solver finds the better configuration for the VAEs for the given and forecasted workload. The solver is also virtualization aware, as it takes into account the time required for virtual machine provisioning, i.e. the time required to activate a virtual machine and the time required for closing it once it's no longer needed; the overhead due to virtualization itself is assumed as one tenth of the available resources.

In [1], the model focuses on SLA violations, trying to minimize them. To get a solvable performance model, the probability of a service time bigger than the agreed value is bounded via the Markov Inequality, and the model is monodimensional.

In [4] we have proposed a multi-dimensional model, that we discuss and extend in this paper.

3. Problem statement

We assume that our infrastructure is a multi-tier system, where each tier has a different size that could change over time due to changing workload; each tier is comprised of elements with the same resources requirements. These elements will be implemented via virtual machines, that have to be allocated over a set of physical nodes. Both virtual machines and physical nodes are described with a multi-dimensional vector (a resources demand vector for virtual machines and a resources available vector for physical nodes). An example of components of these vectors are CPU cores or MHz, memory size (MB), network or I/O bandwidth, but also qualitative resources as physical location, hardware support for virtualization or software licenses could be modeled. The number of dimensions is determined by the desired accuracy of the model: usually CPU power is not sufficient to characterize the performances of a system: a node belonging to the database tier will be more characterized by its bandwidth to the storage subsytem, whilst performances of a node belonging to the web tier are definitely more influenced by the available network bandwidth.

Each of the virtual machines could be described by different SLAs, so we could have a basic version of it and (one or more) premium ones, with progressively increasing demand of resources. The idea behind the different SLAs is that we must allocate the basic version of each virtual machine to have the distributed system working, but then we could try to increase the service level provided to some nodes to maximize physical resources efficiency and accommodate for transient workload surges.

For each virtual machine that we allocate, being a basic or a premium service, we gain some profit. We have two partially conflicting goals: first, we want to maximize the profit we earn from the allocation of the infrastructure, so we'll try to offer premium services instead of basic services whenever they are available; second, we want to minimize the number of physical hosts devoted to hosts the infrastructure. This second goal is modeled by considering an economic penalty proportional to the number of physical hosts used. This penalty accounts for operational costs of the data center like hardware maintenance and support contracts, electricity bills, staff expenses and so on.

In 3.1 we present the proposed formal model, and in 3.2 we evaluate its computational complexity.

3.1. The Formal Model

The virtual machines that comprises our infrastructure have more than one SLA associated with them, so we adopt the convention that if X is a scalar or vector quantity then X^{ij} is the scalar or vector of the machine i with a service level j. We define the set $\{X^{ij}\}$ for a fixed i as the *group* i.

Also, we extend the $\leq$ operator from scalar to vectors in a coordinate wise fashion: if $X = (x_1, x_2, ..., x_n), Y = (y_1, y_2, ..., y_n)$ we say that $X \leq Y$ iff $x_i < y_i$ for each i s.t. $1 \leq i \leq n$.

In our model:

- G is the number of virtual machines. Each one has g_i different service levels (it's possible that $g_i = 1$ for some or even all i's);
- each virtual machine is described by a K-dimensional demand vector $D^{ij} = (d_1^{ij}, d_2^{ij}, ...d_k^{ij})$;
- each SLA of each virtual machine has an associated profit P^{ij};
- for each physical host we use (totally or partially) we pay C units to cover operational costs (hardware, support fees, staff expenses, ...);
- M is the number of physical hosts;
- each host is described by a K-dimensional resource vector:
  $R^l = (r_1^l, r_2^l, ..., r_k^l), 1 \leq l \leq M$;
- for each i, we have $D^{i1} \leq D^{i2} \leq ... \leq D^{ig_i}$, $P^{i1} \leq P^{i2} \leq ... \leq P^{ig_i}$.

In our model, the decision variables x_m^{ij} are set to 1 if the SLA j of the virtual machine i is instantiated over the physical hosts m, otherwise are set to 0.

We want to choose one and one only virtual machine and allocate it on a physical host, with the constraint that we cannot exceed the available resources provided by the hosts, maximizing the total profit earned and minimizing the number of physical machines that are used.

Informally a group could be defined as the set of all the decisional variables that belong to a specific virtual machine: only one variable of the set is non-zero, meaning that we choose only one of the available SLAs to be allocated over a specific physical host. If we have a tier comprised of 10 virtual machines, we will have 10 groups in our problem, one for each virtual machine, and so on for all the tiers.

To do so, we define the variables u^m to have the value 1 if the physical host m has at a least one virtual machine instantiated over it (so it's being used), otherwise 0.

Our constraints are formally defined as:

$$\forall i\, , \sum_{m=1}^{M} \sum_{j=1}^{g_i} x_m^{ij} = 1 \tag{1}$$

$$\forall m\, , \sum_{i} \sum_{j} x_m^{ij} D^{ij} \leq R^m \tag{2}$$

$$\forall m, i, j\, ,\ u_m \geq x_m^{ij} \tag{3}$$

Eq. 1 means that we choose only one virtual machine for each group, and eq. 2 means that, on each physical machine, we cannot allocate more resources than available, while eq. 3 relates variables u_m to x_m^{ij}.

Our objective function is then:

$$P = \sum_{i=1}^{G} \sum_{j=1}^{g_i} \sum_{m=1}^{M} x_m^{ij} P^{ij} - C * \sum_{m=0}^{M} u_m \tag{4}$$

The proposed model allow for coexistence of both quantitative and qualitative resources.

As an example, we might want to spread the distributed system in two different areas (two different LANs, or two geographically remote sites). To do so, we extend the quantitative model, by defining two new qualitative resources, called q_1 and q_2. Resource q_1 means 'allocation in the first area', whilst resource q_2 means 'allocation in the second area'. We extend the demand resource vector to accommodate for the new resources, and we put $q_1 = 1$ for the first half of nodes of the tier, and $q_2 = 1$ for the second half, meaning we want half nodes in the first area and half nodes in the second area. Lastly, for the physical machines located in the first area, we set the available resource $q_1 = M/2$ and the available resource $q_2 = 0$; the converse for the physical hosts located in the second area. As a result, each solution will map half virtual machines (machines for which the demand vector has $q_1 = 1$) on the physical hosts located in the first area, and remaining on physical hosts located in the second area, thus giving us a geographical distribution of the infrastructure.

3.2. Model discussion

This problem is a generalization of the well known knapsack problem [9], because:

- the knapsack problem is mono-dimensional, whilst this is multi-dimensional;
- the knapsack problem has only one knapsack (physical host), whilst this problem deals with multiple knapsacks (physical hosts);
- the knapsack problem doesn't group items, this does (different SLAs for the same virtual machines).

Some minor generalization of the knapsack problem have been studied before. We remand to our work [4] for the discussion. Here we want to stress that our model assumes that the virtualization technology is capable of performing robust and fair resources sharing among all the virtual machines over the same physical node.

To determine the computational complexity of finding the optimal solution of the formal model, we start by observing that, if we consider the set X_{ij}^m for a fixed i, we have exactly one element of it set to 1, where the set is comprised of $g_i * M$ elements. Iterating all over the different sets, we have that the solution space has a size S of:

$$S = \theta(\prod_i^G (g_i * M)) = \theta(\prod_i^G g_i * M^G) \tag{5}$$

When we used the θ notation as not all solutions are feasible. If $\forall i \; g_i = g$, in eq. 5 we get:

$$S = \theta((g * M)^G) \tag{6}$$

Each classical 0/1 knapsack problem could be represented as a specific instance of our problem.

Consider a 0/1 knapsack problem where knapsack capacity is k, item weight are w_i and profits are p_i. First, we can extend this to a multi-dimensional problem by substitution of each item weight w_i and knapsack capacity k (both scalar) with vectors $W_i = (w_i, 0, 0..., 0)$ and $K = (k, 0, 0, ..., 0)$. Then we can add more dummy knapsacks to have a multiple knapsacks generalization, and these knapsacks are described via capacity vectors $K' = (0, 0, ..., 0)$, $K'' = (0, 0, ...0)$. Generalization to have a multiple choice problem could be obtained if, for each item described via a vector W_i, we define two demand vectors, $W_i^0 = (0, 0, ..., 0)$ and $W_i^1 = W_i$, with profits respectively $P^{i0} = 0$ and $P^{i1} = p_i$. As a result of this, the problem we are tackling is NP-hard and we must find an approximate way to solve the formal problem. On our previous work [4] we have developed an heuristic that we briefly outline here:

- We perform a permutation of the elements of the problem, as the next step of the heuristic is really sensitive to element ordering;
- We choose a basic solution, i.e. a solution where we instantiate only the basic service level of each virtual machine, by using a well-known strategy as the first-fit, best-fit or next-fit algorithm [7];
- We improve the solution trying to promote the service level of each virtual machine, as long as there is room for that in the physical host and according to an order determined by a goodness function that is the heart of the heuristic.

This heuristic performs well for small values of C/P^{ij} and it's capable of finding the optimal solution (when we could calculate with an integer programming resolver). When the values of C/P^{ij} become bigger, the relative performances of the heuristic decrease.

4. The Genetic Algorithm

A genetic algorithm could be seen as an intelligent probabilistic search in the space of solutions for an hard problem. Starting from the name itself, the terminology of the genetic algorithms is derived from the evolutionary biology, where individuals stem from a population by a recombination of genetic characteristic of their parents, plus a small probability of some random genetic mutation. Some mutations are for the better, giving the individual a higher chance to become a parent of a new individual (that could inherit this advantageous mutation), other mutations are for the worst, and the individual carrying them will have a smaller chance to become parent. Genetic algorithms have been widely considered as an optimization strategy for hard optimization problems, where it's easy to find some solutions but it's very difficult to find the optimal solutions, because these initial solutions could be the initial population from which to search for the optimal one. In the field of integer programming, the mapping between an individual and a solution is usually really simple, as the i-th chromosome of the individual is 0 (or 1) if and only if the i-th decision variable of the portrayed solution is 0 (or 1). Although more complex representations are possible [12] we choose to stick with this. Genetic algorithm are not a free lunch in the field of optimization when they are applied to a constrained optimization problem, as the result of recombination and mutation of two feasible element (i.e., individuals that represent feasible solutions) could not be feasible. The problem expressed by eqs. 1-4 is particularly complex from this point of view. In fact we have two different sets of constraint, the first requires we choose only one element from each group, the other that we don't overfill a physical host. As these two sets of constraints must be enforced together, we cannot adapt a simple 'repair' operator to deal with unfeasible individual (i.e., individual representing unfeasible solutions), as has been done in [5] where, should a physical host be overfilled, elements are removed from it until the violation is fixed: we cannot do that as we *must* allocate exactly one element from each group. The approach we have adopted is to consider our constraints as belonging to two different sets: easy and hard. An easy constraint is a constraint that, should an individual violate it, we could easily fix, while hard constraints require a complementary approach, based on the use of penalty function.
Formally speaking, if we have this optimization problem:

$$\begin{cases} \max \; f(\mathbf{x}) \\ \mathbf{x} \in E \\ \mathbf{x} \in H \end{cases} \tag{7}$$

where E and H represents respectively easy and hard constraints, we transform problem 7 into this one:

$$\begin{cases} \max\ f(\mathbf{x}) - p(d(\mathbf{x}, H)) \\ \mathbf{x} \in E \end{cases} \tag{8}$$

where $d(\mathbf{x}, H)$ is a metric function describing the distance of solution **x** from the set H of feasible solutions, and $p(\cdot)$ is a monotonically non-decreasing function such that $p(0) = 0$. Penalty functions are surveyed in [2]. For our model, constraint (1) is easy, so we define a repair operator for individuals that violate it, while constraint (2) is hard, and it will be handled via a penalty function [2].

4.1. Outline

Our genetic algorithm starts with a population that is made up of individuals representing basic solution for the problem, i.e. solutions where only the lowest SLA of each virtual machine has been chosen to be allocated. We generate these solutions by using the first-fit, best-fit and next-fit heuristic from the bin-packing problem (for both the two scenarios considered later, we generate 100 unique individuals by each heuristic). The heuristics are strictly deterministic: they all start considering the first virtual machine and choose where it could be allocated, then move on the second virtual machines and so on. In order to generate more different solutions, we change the order in which we process the virtual machines, by doing a random permutation. We must take care that we don't insert into the population an element that already is in it, to avoid that we unnecessarily reduce the initial population size. After this initialization step, we do an iterative process, each cycle of it called a generation, where we:

1. Choose the two parents of the new individual, by a tournament process;
2. Create the new individual by a crossover operator;
3. Mutate some variables of the new individual with a mutation operator;
4. Calculate the fitness of the individual, taking care of unfeasibility due to overfilling;
5. Fix the easy constraint with the repair operator;
6. Insert this individual into the population, and remove the individual with the lowest fitness.

These steps are all tunable by some parameters, resulting in different instances of the same genetic algorithm. We discuss each of these steps in the following paragraphs.

4.2. Tournament Process

To choose the two parents that will generate a new individual, we randomly define two different pools of all different elements from the population. From each pool, we choose the element with the highest fitness as the parent. A larger pool will increase the competitive pressure. Somewhat arbitrarily, we set the pool size to 5.

4.3. Crossover Operator

After the selection of the two parents, the new individual will be defined as the crossover of them. Instead of adopting a random crossover we do an uniform crossover [6], where the probability that the i-th variable of the new individual is equal to the i-th variable of the first or second parent is proportional to the fitness of the first or second parent.

4.4. Mutation Operator

Mutation rate is fixed, and it's chosen as 3 times the relative frequency of the variables set to 1 all over the population. A more complex approach would be a dynamic mutation rate, with higher rate for the initial generations (when we are probably away from the optimal solution, so we can change a lot of variables) and then lowering it as the generations pass away. This is a critical parameter, as an high rate could destroy the stability of the genetic algorithm, and a low rate could end up in being trapped in a local minimum.

4.5. Fitness and penalty function

At a first glance, one should be tempted to consider the objective function (4) as the fitness function, but this will result in even completely different individuals with the same fitness, when we want to differentiate as much as possible in order to pick up, during the tournament process, the potentially best individuals by looking at their fitness and not only by chance. We have also to include the penalty function in the fitness computation, so we are already considering a different problem than the original one, but we must define the fitness function in such a way that individuals with better fitness represent, on average, better solutions for the original problem. We observe that, if we have two different and feasible solutions **x** and **x'** with the same value of the objective function (4) and the same number of physical hosts used, we can still say that **x** is better than **x'** if **x** packs more virtual machines in the same physical host, while **x'** allocate virtual machines more evenly; this is because it's more probable that, from the solution **x** we have more unused resources in some physical hosts and we can use these resources to allocate some others virtual machines, without changing the number of physical hosts used, while for solution **x'** unused resources are not aggregated together. Formally speaking, for a solution **x** of the formal problem, we define the relative amount of unused resources for each physical hosts as:

$$rel_k^m = \frac{r_k^m - \sum_{i=1}^{G} \sum_{j=1}^{g_i} x_k^{ij} * d_k^{ij}}{r_k^m} \tag{9}$$

From this definition, we have that rel_k^l is not negative when the physical host l has some unused resource k, and it's less than zero when we have overfilled it with respect to that resource. We can leverage on this property of rel_k^l by using both for rewarding feasible individuals and for penalizing unfeasible individuals. To do so, we need also to define the portion of the profit on a per physical hosts basis, i.e. the profit we earn for the virtual machines allocated on a specific physical hosts:

$$Gain(m) = \sum x_m^{ij} * P^{ij} \tag{10}$$

we need this value to deal with the unfeasibility that arises after overfilling a physical host. In such a case, we cannot say which virtual machines is 'guilty', and we have to decrease the total profit for the portion of the profit we earn from all the virtual machines allocated over this overfilled physical host (the inability to say which virtual machines is guilty is what makes difficult to define a repair operator and forces use to search a suitable penalty function). Conversely, when the host is not overfilled, we could increase the fitness, and the more resources are relatively free (hosts by hosts), the more we increase the fitness. Putting all together, we define the fitness function as:

$$F(\mathbf{x}) = P(\mathbf{x}) + \sum_{i=1}^{K} \sum_{m=1}^{M} Gain(m) * \frac{\alpha}{K} * rel_k^m \tag{11}$$

The quantity α is used as a static multiplier: if $\frac{\alpha}{K} > 1$ we reward and penalize individuals more aggressively.

4.6. The Repair Operator

We define a repair operator to ensure that eq. 1 is true for each individual. This equation requires that, for each group x_i, we have exactly one element set to 1, all others being 0. We could express in a different way by stating that, for each group x_i: 1) there is at least an element different than 0; 2) there is no more than one element different than 0. By such separation, we can define two specialized operators. Each individual is an ordered collection of groups, and eq. 1 could hold for some groups and not for others. So we scan all the groups comprising the individual to check for property 1, we somehow fix the groups that don't verify it, and then we rescan all the groups to check and possibly fix for property 2.

To describe the process, consider an individual made up of 3 groups (which means that we are searching for the optimal allocation of 3 virtual machines) where first group has no element set to 1, and second and third group both have 2 elements set to 1 (see figure 1 for a pictorial representation).

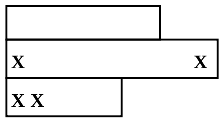

Figure 1. An individual for a problem with 3 groups. Each X marks a variable set to 1.

To ensure that each group has at least one non-zero element, we need to fix the first group. A naive approach would be choosing randomly one element of the first group and setting it to 1, but we prefer performing an intelligent search between the neighbours.We generate g_1 new individuals, each one of them completely equal to the individual we are fixing, but with the i-th variable of the first group set to 1. (see figure 2). For each of these individuals, we evaluate the fitness (our fitness function takes care of unfeasibility, so we can safely use it) and we choose the individual with the highest fitness as the repaired individual. If we have more than one group that needs this fixing, we perform it in an iterative way, group after group.

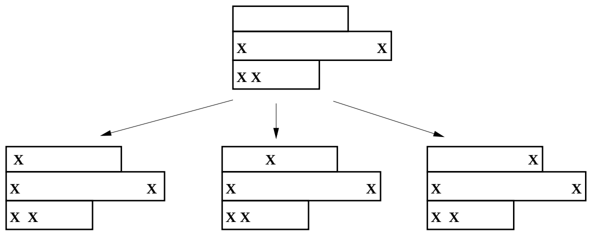

Figure 2. Fixing the first group by generating different individuals.

Now we have to ensure that each group has no more than one non-zero element, so we need to fix the second group. We generate two individuals, where the i-th individual has set to 1 only the i-th non-zero variables of the second group, and again we choose the best among them. Then we repeat the process for the third group (see figure 3).

We stress that, when we create the neighborhood list, we have partially unfeasible individuals in it, but we can cope with this as the fitness function is robust enough. We could use a more complex research when we consider all the possible combinations (see figure 4), but we have chosen not to do this for this first version of the genetic algorithm, as efficiency should be carefully evaluated, especially for the size of the explored set of neighborhood elements.

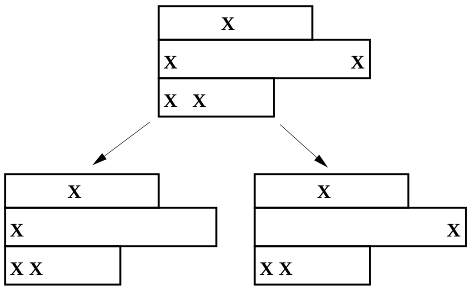

Figure 3. Fixing the second group by generating two different individuals.

5. Experimental Dataset and Results

In the field of the operational research and integer programming, there are some datasets for the most classical problems that one could use to test against a new algorithm, so it's possible to compare the relative performances between different researchers. As of now, we don't have the same for virtualized architectures, and we were unable to find publicly known data depicting a distributed system that has been virtualized: we were forced to consider somewhat arbitrary models of distributed systems, and we are also aware that the best parameters for the genetic algorithm we are considering cannot be determined without knowing the class and the structure of the problems.

Tier	Nodes	CPUs	RAM Size	Profit
Web	4	1/2/4	2/4/6	2/4/8
Application	8	2/4/4	2/4/6	2/4/6
Database	3	2/2	4/6	2/6

Table 1. First model characterization of virtual machines with different SLAs and profits. A notation like *1/2/4* means that we have three different SLAs: the first requires 1 unit of resource (CPU cores), the second 2 units and the third 4 units. Not all the tiers have the same number of SLAs.

Tier	Nodes	CPUs	RAM Size	Profit
Web	6	1/2	2/4	2/4
Application	12	1/4	2/4	2/4
Database	4	2/2	4/6	2/6

Table 2. Second model characterization of virtual machines with different SLAs and profits. Notation is the same of table 1.

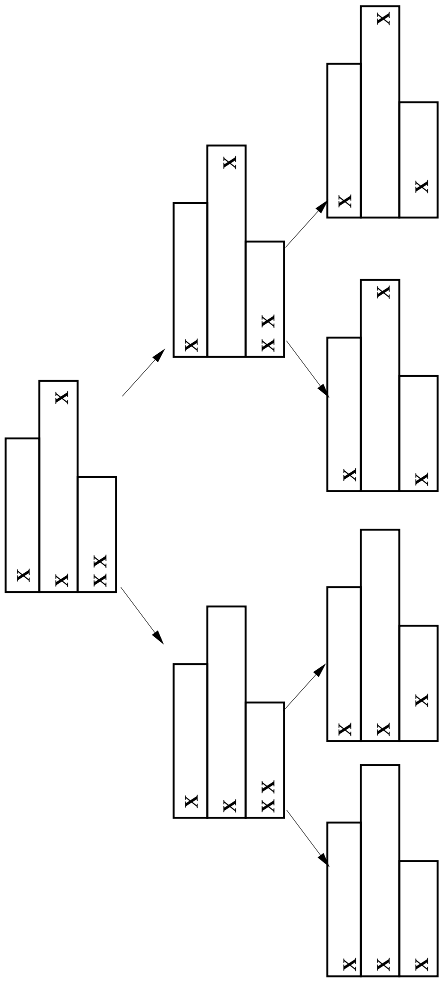

Figure 4. Fixing the second group and third group by generating all possible combinations.

We chose to consider only two different three-tier architecture, reported in tables 1 and 2. For the first scenario, we have 8 physical hosts, 3 of them have 8 CPU cores and 16 GiB of RAM, the others have 8 cores and 8 GiB. In the second scenario, all the 12 physical hosts have 8 CPU cores and 16 GiB of RAM. In both cases, the penalty C for using a machine has been set to 1, so it's always convenient to increase the SLAs for the virtual machines.

We ran the genetic algorithm for 2,000 generations, extracting the average fitness after each one. Figures 5 and 6 show the evolution of the average fitness in the two cases.

Both evolutions are quite similar. Although they start from an initial population of basic solution, the average fitness goes up quickly, and after 300-400 generations the pace of progress is reduces, and this reduction is more pronounced in the second scenario. After 2,000 generations,

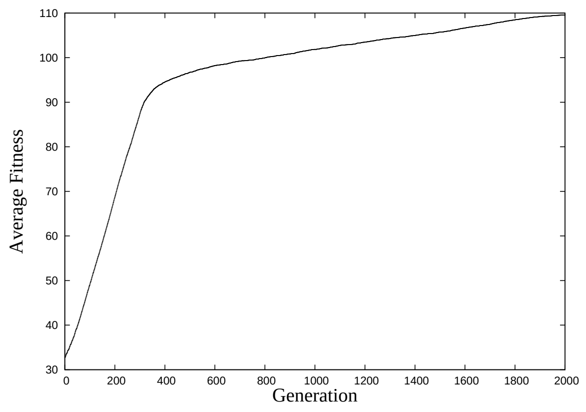

Figure 5. Average fitness for the first scenario.

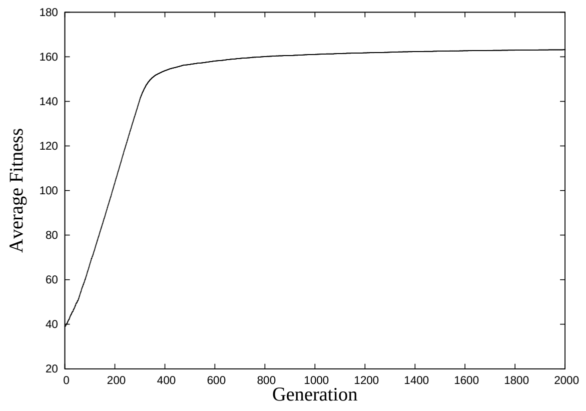

Figure 6. Average fitness for the second scenario.

both populations contain both feasible and not feasible individuals. Table 3 reports, for both scenarios, the initial fitness and the final fitness, which is the highest fitness between all feasible individuals after 2,000 generations have been passed. Note that the initial fitness coincides with the average fitness of the initial population (as the initial population is made up of all feasible individuals), while the final fitness is less than the average fitness of the population at generation 2,000, as many individuals are unfeasible and cannot be considered as a solution. Almost everyone of the 2,000 individual we generate over the course of the genetic algorithm is initially unfeasible, and needs to be fixed with the repair operator. A lot of them also overfill the available physical hosts, but the genetic algorithm has been able to find feasible solution, with a much higher fitness than the original value, which is even more worthy of attention as the original population contains solutions provided by well-known and widely adopted heuristics. We show only these synthetic data as both models have hundreds of different decisional variables and it would be unnecessarily long to show the complete solutions.

Scenario	Initial Fitness	Final Fitness
1	33	109
2	39	166

Table 3. Initial and final fitness for both scenarios.

The time it takes to run the genetic algorithm is well under a minute, over an AMD64 at 3.2 GHz, but we didn't take accurate measurements as the running code wasn't optimized for speed, as we prefer robustness over speed. This efficiency should make the genetic algorithm feasible in a dynamic allocation scenario.

6. Conclusions and future work

Virtualization has come a long way, as its first implementation dates back to '60s, so the resources allocation for a virtualized environment has already been hit by vendors and system architects. Nowadays, virtualization is all the rage, and we could model it by using new methodologies and tools unavailable decades ago, adapting it to a different context.

We have developed a formal model for the allocation of resources in a virtualized environment, and we have developed two different sets of techniques to find approximate solution of it. We are currently looking forward to find structure's characterization of real distributed system to evaluate and tune our genetic algorithm and compare it against our heuristic.

7. Acknowledgments

I'd like to thank the anonymous reviewers for their insightful comments and Ajay Gulati for his support.

References

[1] J. Almeida, V. Almeida, D. Ardagna, C. Francalanci, and M. Trubian. Resource management in the autonomic service-oriented architecture. *Autonomic Computing, International Conference on*, 2006.

[2] T. Baeck, D. Fogel, and E. Z. Michalewicz. *Handbook of Evolutionary Computation*. A jpint Publication of Oxford University Press and Institute of Physics Publishing, 1995.

[3] M. N. Bennani and D. A. Menasce. Resource allocation for autonomic data centers using analytic performance models. In *ICAC '05: Proceedings of the Second International Conference on Automatic Computing*. IEEE Computer Society, 2005.

[4] P. Campegiani and F. L. Presti. A general model for virtual machines resources allocation in multi-tier distributed systems. In *ICAS '09: Proceedings of the International Conference on Autonomic and Autonomous Systems*. IEEE Computer Society, 2009.

[5] P. C. Chu and J. E. Beasley. A genetic algorithm for the multidimensional knapsack problem. *Journal of Heuristics*, 4(1):63–86, 1998.

[6] P. C. C. J. E. Beasley. A genetic algorithm for the set covering problem. *European Journal of Operational Research*, 94:392–404, 1996.

[7] E. G. C. Jr., M. R. Garey, and D. S. Johnson. Approximation algorithms for bin packing: a survey. pages 46–93, 1997.

[8] A. Karve, T. Kimbrel, G. Pacifici, M. Spreitzer, M. Steinder, M. Sviridenko, and A. Tantawi. Dynamic placement for clustered web applications. In *WWW '06: Proceedings of the 15th international conference on World Wide Web*. ACM, 2006.

[9] S. Martello and P. Toth. *Knapsack problems: algorithms and computer implementations*. John Wiley & Sons, Inc., 1990.

[10] D. A. Menasce and M. N. Bennani. Autonomic virtualized environments. In *ICAS '06: Proceedings of the International Conference on Autonomic and Autonomous Systems*. IEEE Computer Society, 2006.

[11] P. Padala, K. G. Shin, X. Zhu, M. Uysal, Z. Wang, S. Singhal, A. Merchant, and K. Salem. Adaptive control of virtualized resources in utility computing environments. In *EuroSys '07: Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*. ACM, 2007.

[12] C. Reeves. Hybrid genetic algorithms for bin-packing and related problems. *Annals of Operations Research*, 63:371–396, 1996.

[13] X. Wang, D. Lan, G. Wang, X. Fang, M. Ye, Y. Chen, and Q. Wang. Appliance-based autonomic provisioning framework for virtualized outsourcing data center. In *ICAC '07: Proceedings of the Fourth International Conference on Autonomic Computing*. IEEE Computer Society, 2007.