Simulated Annealing for the Machine Reassignment Problem

Gabriel Portal, Marcus Ritt, Luciana S. Buriol,
Leonardo Borba, Alexandre Benavides
Instituto de Informática, Universidade Federal do Rio Grande do Sul,
Porto Alegre, Brazil

June 8, 2012

We propose an heuristic solution for the Machine reassignment problem using Simulated Annealing. The method uses two simple neighborhoods. Together with a suitable set of data structures which allow the core operations to run fast, it is able to perform a high number of iterations in short time. A simple but efficient approach is adequate given the size of the instances of the Google/ROADEF challenge and the short time for solving them. We report the results of experiments performed on the set of available instances and compare them to best known values and a lower bound.

1 Introduction

Given a feasible assignment of processes to machines, the Machine Reassignment Problem consists in finding another assignment of the processes to the machines that improves machine usage. The usage of the machines is measured by load, balance and move costs. Moreover, there are several hard constraints that should be considered by the assignment. The full description of the problem is available in [3].

This problem is clearly an NP-hard combinatorial optimization problem, since it includes several NP-complete problems, e.g. bin packing, as special cases. A common decision for solving medium and large instances of such a problem is to use heuristics. We have investigated meta-heuristics based on local search methods applied to this problem. After some initial experiments we decided to use Simulated Annealing (SA) to solve it.

The paper is organized as follows. Section 2 presents the proposed Simulated Annealing. Section 3 describes the data structures. We present a lower bound for the problem in Section 4. The computational results are presented in Section 5, and the paper finishes with the conclusions in Section 6.

2 Simulated Annealing for Machine Reassignment

The proposed Simulated Annealing uses two simple neighborhoods. The first neighborhood moves a task from a machine to another. The second neighborhood exchanges two tasks on different machines. For an instance with p processes and m machines the size of these neighborhoods is $O(pm)$ and $O(p^2)$, respectively. In our method, we decided to combine both neighborhoods. We first choose one of the two neighborhoods, with probability p and 1-p, respec-

tively, and then a neighbor from the chosen neighborhood. We used probability $p = 0.7$ for our computational experiments.

Simulated annealing proceeds by repeatedly selecting a random neighbor of the current solution. An important detail is the choice of a feasible neighbor of the current solution. Ideally, this would be a random choice among the neighbors. Since the neighborhoods can be large, the feasibility test would incur a large time overhead. We therefore opted for a more efficient sampling strategy.

For the first neighborhood, a process and a machine k are selected at random. Then, we consider the machines $(k + i)\%m$ for $0 \leq i \leq c$, where c is a constant (we used $c = 100$). The first valid assignment of the selected process to a machine in the given order, if any, is chosen. Otherwise, the selected process is considered unmovable and the move is rejected. A similar procedure is used for the second neighborhood: a fixed process is chosen and a sequence of c other processes is chosen to perform the movement. This strategy guarantees an efficient choice of the neighbor – constant in the size of the instance.

A cooling cycle of Simulated Annealing starts with an initial temperature t_0, holds the temperature constant for n iterations, and then reduces it with a cooling rate r. When the current best solution value is not updated for $20n$ iterations and the number of accepted moves is less than 0.1% we consider the solution of the current cooling cycle "frozen" [1]. In this case, we reheat by increasing the temperature to $t_0/100$. The objective of this reheating procedure is to perform more significant perturbations to the current solution, hoping to escape of a local minimum.

The optimizer executes in two independent threads with different parameters and returns the best solution found. The discussion about the choice of parameters can be found in Section 5.1.

3 Data Structures

It is important to execute three basic operations fast:

- Verify the feasibility of a move,
- compute the objective cost of a neighboring solution, and
- execute a move.

In order to have fast methods, an increased use of memory was necessary.

For the constraints of resource usage, two matrices machine-resource were necessary for keeping track of actual usage and transient usage of each resource of each machine. For the conflict constraint, a matrix service-machine was used to record if a machine is already being used by a service. The spread constraint needs a matrix service-location, for knowing the number of processes of a service in a given location. An array for determining the current spread of a service was also used. For guaranteeing the dependency constraints, a matrix service-neighborhood keeps track of the number of processes of a given service in each neighborhood. To make the evaluation of the service move cost constant, two arrays were necessary: one maintains the number of moved processes of each service and another the number of services that have a determined number of moved processes.

In summary, for r resources, s services, n neighborhoods, and l locations the above data structures use $O(mr + ms + sl + sn + p)$ memory. This usually will be dominated by the ms term, but the total memory usage is less than 64 MB for all instances according to the problem definition.

The benefit of using more memory is that the number of operations per move is $O(r+d+b)$, where d is the number of dependencies and b is the number of balance costs. In fact, the verification of a move and its execution both cost $O(r+d)$ and the computation of the cost of a neighboring solution costs $O(r+b)$. Since only the dependencies concerning one service are verified, the cost can be expected to be smaller than these upper bounds. The used data-structures allow most of the verifications and updates to be computed in constant time.

4 A Lower Bound

To evaluate the quality of the solutions found, we propose a lower bound for the problem. This was particularly important for the big instances, since our IP formulation for CPLEX did not fit into main memory, and could therefore not even provide a lower bound. The lower bound is the sum of a lower bound for the load cost of the machines and a lower bound for the balance cost. The load cost lower bound is calculated as follows. For each resource, we consider the sum of capacities of machines and the sum of usages of processes. Basically, the constraint of assignment of a process to a single machine is relaxed: the resources of a process may be assigned to many machines and one resource of a process may also be divided in many machines. Let $R(p,r)$ be the requirement of process p for resource r, $SC(m,r)$ be the safety capacity of machine m for resource r and $weight_{loadCost}(r)$ be the weight for the load cost of this resource. Then we have

$$LB1 = \sum_{r \in R} weight_{loadCost}(r) \max(0, \sum_{p \in P} R(p,r) - \sum_{m \in M} SC(m,r)). \tag{1}$$

For the balance cost, the same strategy is used: we calculate the cost of a single-machine relaxation. Let $C(m,r)$ be the capacity of machine m for resource r, $t(b)$ be the target factor of balance b, $r_1(b)$ be the first resource of balance b, $r_2(b)$ be the second resource of balance b and $weight_{balancecost}(b)$ be the weight for this balance cost. Then we have

$$LB2 = \sum_{b \in B} weight_{balancecost}(b) \max(0, t(b)\, E(r_1(b)) - E(r_2(b))), \tag{2}$$

where $E(r)$ is the excess of the total capacity for resource r over the total requirements for this resource, defined as

$$E(R) = \sum_{m \in M} C(m,r) - \sum_{p \in P} R(p,r). \tag{3}$$

It can be easily shown that this is a lower bound on the balance cost by rearranging the definition of the balance cost, given in [3].

The calculated lower bound might not be very strong because it does not take into account move costs and does not consider the combined load cost and balance cost. However, it showed to be very tight for the instances B, since our heuristic found solutions with a relative deviation of at most 5% over the lower bound, except for instance B-3 with relative deviation 37%.

5 Experimental Results

All results have been obtained on a PC with an Intel Core2 Quad CPU Q8200 running at 2.33 GHz and 4 GB of main memory, over a 64 bits Linux operation system (Ubuntu 10.04). The method was implemented in C++ and compiled with the `gcc` compiler, version 4.4.3 with optimization flag -03. The random number generator is the boost implementation of the Mersenne

twister [2]. The data set is composed of two sets (named A and B) with 10 instances each, available by the ROADEF Challenge. Set A was released before the qualification phase, set B was released after the qualification phase.

5.1 Parameter Setting

We systematically tested several combinations of parameter values applied to a subset of the instances. The subset of chosen instances was: A1-4, A2-2, A2-3, A2-5, B-1, and B-3, since we considered these instances to be the most difficult to solve by our method. The values of parameters we tested were the following: $n \in \{10^4, 10^5, 10^6\}$, $r \in \{0.91, 0.95, 0.97\}$ and $t_0 \in \{10^7, 10^8, 10^9\}$. All combinations of these values were tested, and for each parameter setting and instance, we ran five executions with different seeds. With the values of the average for each considered instance, we calculated scores (relative distance of the solution) for each parameter setting. Finally, we ranked the results by the score. The best score was achieved with parameters: $n = 10^5$, $r = 0.97$ and $t_0 = 10^8$.

This parameter setting performed well in the tested instances, but it could be very slow for some instances, spending too many iterations per temperature combined with a slow decrease of temperature. This could be seen specially for instance B-5 (which was not considered in the subset of tested instances), for which some lighter parameter settings performed better. To balance this condition, the parameter setting of the second thread was chosen to be faster than the first one: $n = 70000$, $r = 0.95$ and $t_0 = 10^8$.

5.2 Computational Results

Table 1 presents the results of the proposed method for instances A. For each instance, we report the value of the initial assignment (Initial Value), the best value obtained by the competing teams in the qualification phase (Qualification), the value obtained in 300 seconds by one execution of the optimizer with seed 9999 (SA), and the scores (Score) of the single execution according to the problem specification computed using the qualification results as the best known solutions. This table helps to compare the results of the presented method with the scores obtained by the teams during the qualification phase.

Table 1: Results of the proposed Simulated Annealing on instances A.

Instance	Initial Value	Qualification	SA	Score
A1-1	49528750	44306501	44306935	0.0000
A1-2	1061649570	777532896	777533311	0.0000
A1-3	583662270	583005717	583009439	0.0000
A1-4	632499600	252728589	260693258	0.0126
A1-5	782189690	727578309	727578311	0.0000
A2-1	391189190	198	222	0.0000
A2-2	1876768120	816523983	877905951	0.0327
A2-3	2272487840	1306868761	1380612398	0.0325
A2-4	3223516130	1681353943	1680587608	-0.0002
A2-5	787355300	336170182	310243809	-0.0329

Table 2 presents the results of the method for instances B. In this table the column "Qualification" has been replaced by the column "Lower bound", which gives the lower bound as presented in Section 4. Here, the column "Score" was calculated using the lower bound as the

best known value. Additionally, we provide the relative deviation over the lower bound (column "Dev.").

Table 2: Results of the proposed Simulated Annealing on instances B.

Instance	Initial Value	Lower Bound	SA	Score	Dev. [%]
B-1	7644173180	3290754940	3455971935	0.0216	5.02
B-2	5181493830	1015153860	1015763028	0.0001	0.06
B-3	6336834660	156631070	215060097	0.0092	37.30
B-4	9209576380	4677767120	4677985338	0.0000	0.00
B-5	12426813010	922858550	923299310	0.0000	0.05
B-6	12749861240	9525841820	9525861951	0.0000	0.00
B-7	37946901700	14833996360	14836763304	0.0001	0.02
B-8	14068207250	1214153440	1214563084	0.0000	0.03
B-9	23234641520	15885369400	15886083835	0.0000	0.00
B-10	42220868760	18048006980	18049089128	0.0000	0.01

The method seems to be very competitive, being an improvement of our former method, which already proved to produce good quality results in the qualification phase of the challenge. It produces good results for instances A, finding some new best known values – negative scores. Finally, the method appears to behave robustly in very large instances, as the results on instances B show, since the scores are consistently low. Only two instances with values being 5% and 37% above the lower bound resulted in slightly higher scores.

6 Conclusions

We have proposed an heuristic method based on Simulated Annealing for the Machine Reassignment Problem and a lower bound for it. The Simulated Annealing uses two simple neighborhoods. The results show that the method was able to find near optimal solutions. We believe that the good quality of the results is due to an optimized implementation of the Simulated Annealing, using data structures that allow the most important operations to execute fast and then a larger amount of the solution space can be explored.

References

[1] D. S. Johnson, C. R. Aragon, L. A. McGeoch, and C. Schevon. Optimization by Simulated Annealing. Part I, Graph Partitioning. *Operations Research*, 37:865–892, 1989.

[2] M. Matsumoto and T. Nishimura. Mersenne twister: A 623-dimensionally equidistributed uniform pseudo-random number generator. *ACMTMCS: ACM Transactions on Modeling and Computer Simulation*, 8:3–30, 1998. `http://www.math.keio.ac.jp/~matumoto/emt.html`.

[3] Google ROADEF/EURO challenge 2011–2012: Machine Reassignment. `http://challenge.roadef.org/2012/files/problem_definition_v1.pdf`, 2011. Version 1.