ROADEF/EURO Challenge 2012: Team J33

A. Sansottera, L. Ferruci, N. Calcavecchia, F. Sironi

Dipartimento di Elettronica e Informazione
Politecnico di Milano
Via Ponzio 34/5
20133 Milano, Italy
{sansottera,ferrucci,calcavecchia,sironi}@elet.polimi.it

I. Algorithm Description

One of the first decisions that we made during the development of our program was how to exploit the parallelism of the target machine. Rather than focusing on implementing parallel versions of the heuristics, we decided to develop a flexible framework to run multiple heuristics in parallel. This strategy was very useful in encouraging experimentation with different heuristics. Our program can start multiple heuristics simultaneously to exploit multiple CPU cores. Instances of different heuristics can run together. Moreover, multiple instances of the same heuristic can be run with different parameters. The heuristics to run and their parameters can be specified at the command line, hence simplifying the test of multiple heuristic combinations and parameter configurations.

Communication between heuristic instances happens through a shared solution pool (class `SolutionPool`). The solution pool provides thread-safe access to two separate sets of solutions: high-quality and high-diversity solutions. The set of high-quality solutions contains the 100 solutions with the best objective value among the ones distant at least two moves from the best solution. This condition avoids keeping multiple solutions from a small neighborhood of the best solution. The set of high-diversity solutions contains the 20 solutions with a good enough objective value, that are the most distant from the best solution. The condition on the objective value avoids bad solution from entering the high-diversity pool. Both pools are rebuilt when a knew best solution is found. The solution pool also supports notifications: heuristics can wait for new solutions to enter the pool. This possibility is exploited by our path relinking implementation.

During the course of the competition, we implemented several heuristics, including variable neighborhood search [1], simulated annealing [2], tabu search [3] and path relinking [4]. Moreover, for each of these heuristics, we tested several variants and parameter configurations. For instance, we tested both first-improvement (hill climbing) and best-improvement (steepest ascent) strategies in the local search routine of our variable neighborhood search implementation.

After several experiments on the problem instances of set B, we decided to run an instance of our variable neighborhood search [1] implementation and one instance of our simulated annealing implementation. Details about the two algorithms are provided in Section IV and Section III, respectively.

II. Neighborhoods

In our local search heuristics, we consider the k-neighborhoods reachable by k steps of two different types:

- *move*: pick a process p and a machine $m_{\text{dst}} \neq m_{\text{src}}$ and move process p to machine m_{dst};
- *exchange*: pick two processes p_1 and p_2 assigned to machines m_1 and $m_2 \neq m_1$, respectively, and move p_1 to m_2 and p_2 to m_1.

In order to quickly evaluate feasibility and objective cost, we implemented the class `SolutionInfo`, whose instances contain the following data:

- the solution vector, i.e. the assignment of processes to machines;
- the usage of resource r at machine m, for any $(r, m) \in \mathcal{R} \times \mathcal{M}$;
- the transient usage of resource r at machine m, for any $(r, m) \in \mathcal{R} \times \mathcal{M}$;
- the number of processes of service s on machine m, for any $(s, m) \in \mathcal{S} \times \mathcal{M}$;
- the number of processes of service s in location l, for any $(s, l) \in \mathcal{S} \times \mathcal{L}$
- the number of processes of service s in neighborhood n, for any $(s, n) \in \mathcal{S} \times \mathcal{N}$
- the spread of service s, i.e. the number of different locations in which the service is present, , for any $s \in \mathcal{S}$;
- the load costs, balance costs and movement costs.

These data structures are read and modified by the `MoveVerifier` and `ExchangeVerifier` classes. These two classes always make the assumption that the current solution is feasible and are responsible to

- evaluate if the move (or exchange) is feasible;
- evaluate the updated objective cost resulting from the move (or exchange);
- apply the move (or exchange) by updating a `SolutionInfo` object.

A lot of effort was spent in optimizing the code in `MoveVerifier` and `ExchangeVerifier`. In particular, special handling of services with a single processes and services with no dependency was crucial in reducing the number of misses in the last level cache. Moreover, we implemented

an optimization to compact the machine move cost matrix to 8 bit integers (instead of a 32 bit integers) when they are sufficient to represent all the cost values (see the class `Problem`). This optimization alone reduced by about 10% the number of clock cycles with outstanding last-level cache misses, hence improving the time required to compute the updated objective cost. The `BatchVerifier` class can also evaluate moves (but not exchanges) and does not assume the current solution to be feasible. Furthermore, it keeps track of violated constraints. Due to computational inefficiencies, this class, which was used in the qualification phase, went unused. In fact, we observed that exchanges partially compensates the impossibility to perform unfeasible moves in the local search routines.

III. Simulated Annealing

Our simulated annealing implementation (class `SimulatedAnnealing`) is a modification of the classical simulated annealing meta-heuristic [2]. The main differences with the classical algorithm are based on the work [5].

In particular, the initial temperature T_I is computed with an initial local search and it is set as the maximum difference Δ_{MAX} between the value of the objective function of the initial solution and the value of the objective function of the solutions found with the local search. We define a neighborhood $\mathcal{N}_k(x)$ of a solution $x \in \mathcal{X}$ as the set of solutions $x' \in \mathcal{X}$ such that x' differs form x by a single process $movement$ or an $exchange$, as defined in II. Given a problem instance with $|\mathcal{M}|$ number of machines and $|\mathcal{N}|$ processes, the size of the neighborhood set is fixed to $|\mathcal{M}| * (\log|\mathcal{M}| + \log|\mathcal{N}|)$. This value represents tradeoff between the computation time and the accuracy of the estimation.

The heuristic calls internally a specialized local search (class *SALocalSearchRoutine*), that is terminated when a certain condition $TCond$ holds. The local search tries to find the solution with the best objective function value by exploring a certain number of neighborhood solutions, performing a $movement$ instead of an $exchange$ with a probability p_{move} that is proportional to the current temperature. The probability always varies between two pre-computed constant $MaxProb$ and $MinProb$, defined in class `SALocalSearchRoutine`. At high temperatures, the probability to make a $movement$ is higher than to make an $exchange$ and decrease smoothly with the formula:

$$p_{move} = MinProb \cdot \frac{(\log T_{cur} - \log T_{min})}{(\log T_{max} - \log T_{min})}$$

where T_{cur} is the current temperature. The formula is computed when the condition $(Prob \leq MinProbMove)$ holds.

Our version of simulated annealing also includes a way to reset the temperature whenever the solver might be stuck in a local optima for too many iterations. In particular, we reset the temperature to the value T_{rst} whenever the local search did not find a good local move (movement or exchanges) for $violation_threshold$ times. Moreover the resetting temperature T_{rst} is halved whenever a reset occurs. This allow to exit from the local minima without taking significantly worse solutions. Since the reset can occur many times during the iterations, we also bound its value to a fixed minimum.

Figure 1 depicts the trend of the temperature followed by the simulated annealing in instance b2. As it is possible to notice, the temperature presents some spikes corresponding to the temperature reset action. Each time the temperature is reset, its starting value is smaller than the previous reset value. This characteristic allows to move out from the current local minima in relation considering also the goodness of the current simulation.

Finally, whenever the a new iteration of the simulated annealing is started, the current best solution is pushed into the shared solution pool so that other heuristics can take advantage of progresses made by the simulated annealing.

IV. Variable Neighborhood Search

Our implementation of variable neighborhood search (class `VNS3`) alternates invocations of a shake routine and a local search routine. Several variants of these two routines were implemented. The variants that yield the best performance are described below. Generally speaking, a shake routine jumps to a solution in the k-neighborhood of the best known solution, i.e. to a solution reachable with k moves or exchanges from the best known solution. The local search routine, on the other hand, operates on 1-neighborhoods. If the local search routine does not improve over the current best known objective, the value of k, which ranges from *kMin*=1 to *kMax*=100 is increased by *kStep*=1. If k reaches the value *kMax*, k is reset to *kMin*.

A. Shake Routine

We tested two different shake routines. The first shake routine (class `RandomShakeRoutine`) performs k random feasible steps. Each step can be either a move or an exchange, with equal probability. The steps are accepted regardless of their objective values. If more than a fixed number of trials (the default value is 1000) result in unfeasible moves or exchanges, the shake routine stops before completing and less than k steps are performed. We also implemented a second shake routine (class `DeepShakeRoutine`), which performs a fixed number (parameter *samples*) of k-step sequences steps and selects the one resulting in the best solution. However, the `RandomShakeRoutine` was found to yield the best performance. This shows that diversification is more important than intensification, during the perturbation phase of variable neighborhood search.

B. Local Search Routine

We tested several local search routines. All the routines tested try moves and exchanges with equal probability, which in our tests provided the best overall results. The first routine we tried is a first improvement (hill climbing) method, implemented in class `RandomLocalSearchRoutine`. The performance of this method, however, is poor, since the time to

converge to a good solution was too high. Hence, we focused on best improvement (steepest ascent) methods. We have several implementation and the most efficient one is in class `OptimizedLocalSearchRoutine`. Since evaluating all the possible moves (up to 50000×5000) and exchanges (up to 50000^2) is too computationally expensive, we set a maximum number of trials (parameter *maxTrials*) equal to

$$|\mathcal{P}| \left(\log_{10}(|\mathcal{P}|) + \log_{10}(|\mathcal{M}|)\right) \ . \tag{1}$$

We also set a maximum number of samples, i.e. a maximum number of evaluates moves or exchanges which are feasible and lead to an improvement with respect to the incumbent solution. This is a very important parameter because it marks a trade-off between a first improvement and steepest ascent strategy. After several tests, the parameter *maxSamples* was set to 10000, which seems to yield good result on all the problem instances in set B. The implementation in class `OptimizedLocalSearchRoutine` differs from the other implementations because it is optimized for code locality. Rather than trying random and unrelated moves (or exchanges) at each trial, this routine fills a vector of random processes and machines (or two vectors of random processes for exchanges) and tries all possible moves between these processes and machines (or all possible exchanges between the processes in the two vectors). Hence, the cost of fetching machine and process information in main memory is amortized over several uses, hence improving temporal and spatial locality. The size of this vector is determined by the parameter *block*, that we set equal to 20. Once all possible combinations are tested, new vectors are filled with random process and machine indices.

V. Computation of lower bounds

In order to evaluate our heuristics we found a simple formula to compute a lower bound on the solution cost, which can be quickly evaluated at runtime. However, the current implementation of the heuristics does not leverage this information. We observe that the total solution cost is made of the following components:

- the load cost of each resource r;
- the balance cost of each balance objective b;
- the total movement cost (including process, machine and service movement costs).

A lower bound on the load cost of resource r can be computed by considering a related problem instance, with a single machine whose safety capacity is the sum of the safety capacities of all machines and a single process whose requirement is the sum of the requirements of all processes. The load cost in this related problem instance is computed trivially:

$$\text{LoadCost}_{\text{LB}}(r) = \max\left(0, \sum_{r \in \mathcal{R}} R(p,r) - \sum_{m \in \mathcal{M}} SC(m,r)\right) \ . \tag{2}$$

Intuitively, the load cost in the original problem can never go below this value because, no matter how good is the process assignment, if the total requirement exceeds the total safety capacity, we have to pay for the requirement in excess. A lower bound on the cost of balance objective b can be computed analogously, considering the same problem instance. Let $\Delta(r) = \sum_{m \in \mathcal{M}} C(m,r) - \sum_{p \in \mathcal{P}} R(p,r)$ be the gap between the total capacity and the total requirement of resource r. The lower bound on the balance cost is computed as follows:

$$\text{BalanceCost}_{\text{LB}}(b) = \max\left(0, t\Delta(r_1) - \Delta(r_2)\right) \ . \tag{3}$$

A trivial lower bound on the movement cost is 0, i.e. no process is moved from its initial assignment. Hence, a lower bound on the objective function is

$$\sum_{r \in \mathcal{R}} \text{LoadCost}_{\text{LB}}(r) + \sum_{b \in \mathcal{B}} \text{BalanceCost}_{\text{LB}}(b) \ . \tag{4}$$

While we do not expect this bound to be tight for general problem instances, it works quite effectively for the problem instances in set B.

VI. Computational Results

In this section we present results obtained by our heuristic on the 10 problem instances of set B. Tests were run on a desktop computer with an Intel Core i7 920 processor and 12 GB of DDR3 RAM, running Ubuntu 10.10 64-bit. Since the test machine used by the organizers has a dual-core processor, processor affinity was set to use only two cores of the Core i7 processor.

Results obtained in 300 seconds are shown in Table I. For each problem instance in set B we report the initial objective value, the lower bound computed as described in V and the objective value obtained by our program. Moreover, the table shows the score computed according to the challenge rules, but considering the lower bound as the best solution. Finally, we report the optimality gap computed with respect to the lower bound. The highest impact on the score is due to the first instance, which, despite being one of the smallest, turns out to be the hardest to optimize. For this instance the optimality gap is 2.54%. However, it is hard to say whether this is due to a loose lower bound or if the objective value can be significantly improved. On the other instances we always achieve an optimality gap smaller than 0.70% and the average optimality gap is 0.35%. In Table II, we report the results obtained by increasing the time limit to 1200 seconds. For each instance we report the objective value obtained by our program, the score with respect to the lower bound, the optimality gap with respect to the lower bound and, finally, the relative improvement with respect to the result obtained in 300 seconds. While the improvements are quite small (on average, 0.06%), they show that our solver does not get stuck in a local minima but keeps improving the solution after the 300 seconds time limit, even if they are already quite close to the lower bound. In fact, the total score drops from 1.175 to 1.042 and the average optimality gap drops from 0.35% to 0.29%.

References

[1] P. Hansen and N. Mladenovic, "Variable neighborhood search: Principles and applications," *European Journal of Operational Research*, vol. 130, no. 3, pp. 449 – 467, 2001. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0377221700001004

Instance	Initial	Lower Bound (LB)	Objective	Score (based on LB)	Optimality Gap (based on LB)
b-1	7,644,173,180	3,290,754,940	3,374,420,568	1.095	2.54%
b-2	5,181,493,830	1,015,153,860	1,016,699,248	0.030	0.15%
b-3	6,336,834,660	156,631,070	157,729,183	0.017	0.70%
b-4	9,209,576,380	4,677,767,120	4,677,890,968	0.001	0.00%
b-5	12,426,813,010	922,858,550	923,016,688	0.001	0.02%
b-6	12,749,861,240	9,525,841,820	9,525,860,309	0.000	0.00%
b-7	37,946,901,700	14,833,996,360	14,843,395,915	0.025	0.06%
b-8	14,068,207,250	1,214,153,440	1,214,440,127	0.002	0.02%
b-9	23,234,641,520	15,885,369,400	15,885,531,298	0.000	0.00%
b-10	42,220,868,760	18,048,006,980	18,049,314,317	0.003	0.01%
Total Score				1.175	
Average					0.35%

TABLE I
COMPUTATIONAL RESULTS OBTAINED IN 300 SECONDS ON A CORE I7 920 PROCESSOR.

Instance	Objective (1200 s)	Score (based on LB)	Optimality Gap (based on LB)	Improvement from 300 s to 1200 s
b-1	3,367,279,522	1.001	2.33%	0.21%
b-2	1,015,834,609	0.013	0.07%	0.09%
b-3	157,341,438	0.011	0.45%	0.25%
b-4	4,677,890,968	0.001	0.00%	0.00%
b-5	922,988,809	0.001	0.01%	0.00%
b-6	9,525,858,324	0.000	0.00%	0.00%
b-7	14,838,072,831	0.011	0.03%	0.04%
b-8	1,214,419,057	0.002	0.02%	0.00%
b-9	15,885,528,383	0.001	0.00%	0.00%
b-10	18,048,489,888	0.001	0.00%	0.00%
Total Score		1.042		
Average			0.29%	0.06%

TABLE II
COMPUTATIONAL RESULTS OBTAINED IN 1200 SECONDS ON A CORE I7 920 PROCESSOR.

[2] S. Kirkpatrick, C. D. G. Jr., and M. P. Vecchi, "Optimization by simulated annealing," *Science*, vol. 220, no. 4598, pp. 671–680.

[3] F. Glover and M. Laguna, *Tabu Search*. Norwell, MA, USA: Kluwer Academic Publishers, 1997.

[4] C. Ribeiro and M. Resende, "Path-relinking intensification methods for stochastic local search algorithms," *Journal of Heuristics*, pp. 1–22, 10.1007/s10732-011-9167-1. [Online]. Available: http://dx.doi.org/10.1007/s10732-011-9167-1

[5] I. H. Osman and N. Christofides, "Capacitated clustering problems by hybrid simulated annealing and tabu search," *International Transactions in Operational Research*, vol. 1, no. 3, pp. 317–336.

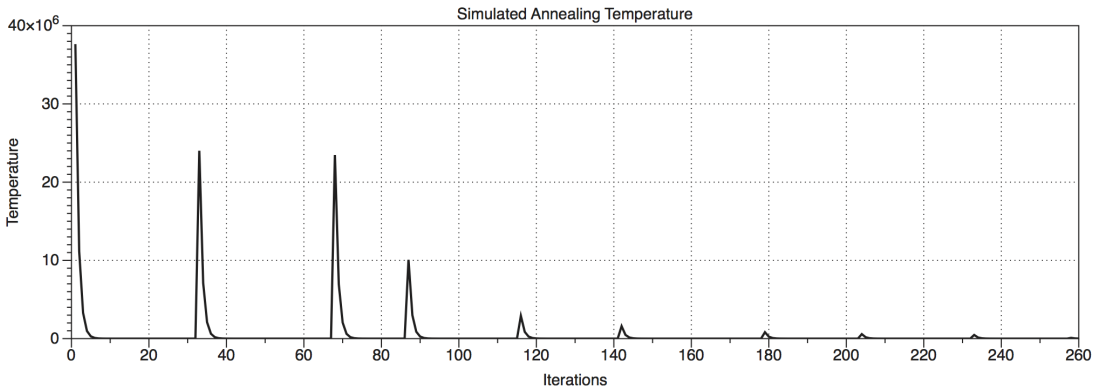

Fig. 1. Trend followed by the temperature in the simulated annealing heuristic