2

On-Line Load Balancing

Without proper scheduling and resource allocation, large queues at each processing operation cause an imbalanced production system: some machines are overloaded while some are starved. With this perspective, the mean cycle time will rise due to local starvation even though the total system inventory stays approximately the same, and it will rise due to the waiting time in the queue for those overloaded processing machines. Hence, the goal is to balance the production facilities to reduce work in progress variability. Thus, production managers are asked to find an allocation strategy so that the workload among the production facilities is distributed as fairly as possible. In other words, managers would like the workload of the different facilities to be kept as balanced as possible. This problem is known as load balancing and is the main topic of this chapter.

2.1 Problem Definition

Formally, the load balancing problem is defined as the problem of the on-line assignment of tasks to n machines; the assignment has to be made immediately upon the arrival of the task, increasing the load on the machine the task has been assigned to for the duration of the task. We consider only nonpreemptive load balancing; *i.e.*, the reassignment of tasks is not allowed. The goal is to minimize the maximum load, or, equivalently, to minimize the sum of the absolute values of the difference between the load of each machine and the average load of the system.

Two main analyzes can be done in this direction: the first one is related to the so called "temporary tasks" problem and the other is the so called "permanent tasks" problem.

The first problem refers to the case in which tasks have a finite duration, *i.e.*, during task arrivals one can observe also task departures from the ma-

chines. In the latter problem, tasks are "permanently assigned" to machines, *i.e.*, only task arrivals are observed over time. This can also be interpreted as a problem in which the task duration is very large with respect to the time horizon where tasks arrive over time.

The other type of distinction can be made according to the duration of the task. In fact, we can have either known duration scenarios or unknown duration scenarios.

The on-line load balancing problem naturally arises in many applications involving the allocation of resources. In particular, many cases that are usually cited as applications for bin packing become load balancing problems when one removes the assumption that the items, once "stored", remain in the storage forever. As a simple concrete example, consider the case where each "machine" represents a plant and a task is a work order. The problem is to assign each incoming order to one of the plants which is able to process that work order. Assigning an order to a plant increases the load on this plant, *i.e.*, it increments the percentage of the used capacity. The load is increased for the duration of the request. Formally, each arriving task j has an associated load vector, $p_j = \{p_{1j}, p_{2j}, \ldots, p_{nj}\}$, where p_{ij} defines the increase in the load of machine i if we were to assign task j to it. This increase in load occurs for the duration d_j of the task.

We are interested in the algorithmic aspect of this problem. Since the arriving tasks have to be assigned without knowledge of the future tasks, it is natural to evaluate performance in terms of the competitive ratio. For this problem, the competitive ratio is the supremum, over all possible input sequences, of the maximum (over time and over machines) load achieved by the on-line algorithm to the maximum load achieved by the optimal off-line algorithm. The competitive ratio may depend, in general, on the number of machines n, which is usually fixed, and should not depend on the number of tasks that may be arbitrarily large. Similar to the way scheduling problems are characterized, load balancing problems can be categorized according to the properties of the load vectors.

2.2 Known Results and Existing Approaches

On-line load balancing has been widely investigated in terms of approximation algorithms (e.g., see [14, 16, 17, 18, 19, 20, 21, 115, 144]). The simplest case is where the coordinates of each load vector are equal to some value that depends only on the task. It is easy to observe that Graham algorithm [80], applied to this kind of load-balancing problem, leads to a $(2\text{-}(1/n))$-competitive solution. Azar *et al.* [17] proposed studying a less restricted case, motivated by the problem of the on-line assignment of network nodes to gateways (see

also [21]). In this case, a task can represent the request of a network node to be assigned to a gateway; machines represent gateways. Since, in general, each node can be served by only one subset of gateways, this leads to a situation where each coordinate of a load vector is either ∞ or equal to a given value that depends only on the task. In this case, which we will refer to as the assignment restriction case, the paper in [17] shows an $\Omega(\sqrt{n})$ lower bound on the competitive ratio of any load balancing algorithm that deals with the unknown duration case, *i.e.*, the case where the duration of a task becomes known only upon its termination. The same authors also present an $O(n^{2/3})$-competitive algorithm. The work in [17] opens several new research ideas. The first is the question of whether there exists an $O(\sqrt{n})$-competitive algorithm for the assignment restriction case when the duration of a task becomes known only upon its termination. Secondly, the $\Omega(\sqrt{n})$ lower bound for the competitive ratio in [17] suggests considering natural variations of the problem for which this lower bound does not apply. One such candidate, considered in this chapter, is the known duration case, where the duration of each task is known upon its arrival.

All the results in this chapter, as well as in the papers mentioned above, concentrate on nonpreemptive load balancing; *i.e.*, the reassignment of tasks is not allowed. Another very different model, is when reassignment of existing tasks is allowed. For the case where the coordinates of the load vector are restricted to 1 or ∞, and a task duration is not known upon its arrival, Phillips and Westbrook [144] proposed an algorithm that achieves an $O(logn)$ competitive ratio with respect to load while making $O(1)$ amortized reassignments per job. The general case was later considered in [16], where an $O(logn)$- competitive algorithm was designed with respect to a load that reroutes each circuit at most $O(logn)$ times. Finally, we note that the load balancing problem is different from the classical scheduling problem of minimizing the makespan of an on-line sequence of tasks with known running times see [81, 157] for a survey. Intuitively, in the load balancing context, the notion of makespan corresponds to maximum load, and there is a new, orthogonal notion of time. See [14] for further discussion on the differences.

2.2.1 The Greedy Approach

When speaking about on-line algorithms, the first that come to mind is the greedy approach. In fact, it is straightforward to allocate an incoming task to the lightest machine. In more detail, one can do as follows:

1. Let $f(s^{(t)})$ be a function defining the maximum load over all the machines at time t, *i.e.*, it returns the weight of the heaviest machine in solution $s^{(t)}$.

2. When a task arrives, let $N(s^{(t)})$ be the neighborhood of current solution $s^{(t)}$;
3. Choose the best solution in $N(s^{(t)})$, *i.e.*, the one that produces the smallest increase of $f(s^{(t-1)})$.

Note that in Line 3 there could be many ties when there is more than one machine for which either the incoming task does not produce any increase in $f(s^{(t-1)})$, or the same smallest positive increment in $f(s^{(t-1)})$ is achieved. In this case one can choose at random the machine in the restricted neighborhood formed exclusively by these solutions.

The greedy algorithm is also known as the Slow-Fit algorithm; it is essentially identical to the algorithm of Aspnes, Azar, Fiat, Plotkin and Waarts for assigning permanent tasks [14]. Roughly speaking, the idea (which originated in the paper by Shmoys, Wein, and Williamson [157]) is to assign the task to the least capable machine while maintaining that the load does not exceed the currently set goal. However, the analysis in [14] is inapplicable for the case where tasks have limited duration. It is known that Slow-Fit is 5-competitive if the maximum load is known. The Slow-Fit algorithm is deterministic and runs in $O(n)$ time per task assignment.

Although this is the simplest way to proceed, both in terms of implementation and in terms of computing times, the lack of knowledge about future incoming tasks can produce the effect of obtaining good solutions in the first algorithm iterations, but may return very low quality solutions after these initial stages.

Thus, one can improve the greedy algorithm by using semi-greedy strategies. At each construction iteration, the choice of where the incoming task has to be allocated is determined by ordering all the candidate elements (*i.e.*, the machines) in a candidate list C with respect to a greedy function $g : C \to R$. This function measures the (myopic) benefit of selecting such an element. In our case g corresponds to f and C contains all the solutions in the neighborhood. Thus, contrary to what the greedy algorithm does, *i.e.*, select the first element in list C which locally minimizes the objective function, the semi-greedy algorithm randomly selects an element from the sublist of C formed by the first r elements, where r is a parameter ranging from 1 to C. This sublist, formed by the first r elements of C, is denoted as the Restricted Candidate List (RCL).

It is easy to verify that if $r = 1$ then the semi-greedy algorithm becomes a greedy algorithm, and if $r = |C|$ then the semi-greedy algorithm behaves like a random algorithm, *i.e.*, a machine is chosen at random and the incoming task is allocated to this machine regardless of how large the increase in f becomes.

For the construction of the RCL, considering the problem of minimizing the maximum load over all the machines at time t, we denote as $\Delta(s_i^{(t)})$ the incremental load associated with allocating the incoming task to machine i, *i.e.*, with the solution $s_i^{(t)} \in N(s^{(t-1)})$ under construction. Let Δ_{min} and Δ_{max} be, respectively, the smallest and the largest increment.

The restricted candidate list RCL is made up of elements $s_i^{(t)}$ with the best (*i.e.*, the smallest) incremental costs $\Delta(s_i^{(t)})$. This list can be limited either by the number of elements (cardinality-based) or by their quality (value-based).

In the first case, it is made up of the r elements with the best incremental costs, where r is the aforementioned parameter. In the latter case, we can construct the RCL considering the solutions in the neighborhood whose quality is superior to a threshold value, *i.e.*, $[\Delta_{min}; \Delta_{min} + \alpha(\Delta_{max} - \Delta_{min})]$, where $0 \leq \alpha \leq 1$. If $\alpha = 0$, then we have a greedy algorithm, while if $\alpha = 1$, we obtain a pure random algorithm.

2.2.2 The Robin-Hood Algorithm

In this section we describe a $(2\sqrt{n}+1)$-competitive algorithm that also works with assignment restriction, where the task duration is unknown upon its arrival. Task j must be assigned to one of the machines in a set M_j; assigning this task to a machine i raises the load on machine i by w_j. The input sequence consists of task arrival and task departure events. Since the state of the system changes only as a result of one of these events, the event numbers can serve as time units; *i.e.* we can view time as being discrete. We say that time t corresponds to the t-th event. Initially, the time is 0, and time 1 is the time at which the first task arrives. Whenever we speak about the "state of the system at time t", we mean the state of the system after the t-th event is handled. In other words, the response to the t-th event takes the system from the state at t-1 to the state at t. Let OPT denote the load achievable by an optimum off-line algorithm. Let $l_i(t)$ denote the load on machine i at time t, *i.e.*, after the t-th event. At any time t, we maintain an estimate $L(t)$ for OPT satisfying $L(t) \leq OPT$. A machine i is said to be rich at some point in time t if $l_i(t) \geq \sqrt{n}L(t)$, and is said to be poor otherwise. A machine may alternate between being rich and poor over time. If i is rich at t, its windfall time at t is the last moment in time at which it became rich. More precisely, i has windfall time t_0 at t if i is poor at time $t_0 - 1$, and is rich for all times t' where $t_0 \leq t' \leq t$.

The Robin-Hood Algorithm.

The Robin-Hood (RH) algorithm is simple, deterministic, and runs in $O(n)$ time per task assignment. Interestingly, its decision regarding where to assign

a new task depends not only on the current load on each machine, but also on the history of previous assignments [18, 20].

Assign the first task to an arbitrary machine, and set $L(1)$ to the weight of the first task. When a new task j arrives at time t, set:

$$L(t) = \max\{L(t-1), w_j, \mu(t)\}$$

The latter quantity, $\mu(t)$, is the aggregate weight of the tasks currently active in the system divided by the number of machines, *i.e.*, $\mu(t) = \frac{1}{n}(w_j + \sum_i l_i(t-1))$. Note that the recomputation of $L(t)$ may cause some rich machines to be reclassified as poor machines.

The generic steps of the RH algorithm are the following:

1. If possible, assign j to some poor machine i.
2. Otherwise, j is assigned to the rich machine i with i being the most recent windfall time.

Lemma 1. *At all times t, the algorithm guarantees that $L(t) \leq OPT$.*

Proof. The proof is immediate since all three quantities used to compute $L(t)$ are less than or equal to OPT.

The following lemma is immediate since $nL(t)$ is an upper bound on the aggregate load of the currently active tasks.

Lemma 2. *At most $\sqrt{n}$ machines can be rich at any point in time.*

Theorem 1. *The competitive ratio of the RH algorithm is at most $2\sqrt{n}+1$.*

Proof. We will show that the algorithm guarantees that at any point in time t, $l_i(t) \leq \sqrt{n}L(t) + OPT) + OPT$ for any machine i. The claim is immediate if i is poor at t. Otherwise, let S be the set of still active tasks at time t that was assigned to i since its windfall time t. Let j be some task in S. Since i is rich throughout the time interval $[t_0, t]$, all the machines M_j that could have been used by the off-line algorithm for j must be rich when j arrives. Moreover, each machine in $M(j) - \{i\}$ must have already been rich before time t, since otherwise, the RH algorithm would have assigned j to it. Let k be the number of machines to which any of the tasks in S could have been assigned; *i.e.*, $k = |\cup_{j \in S} M_j|$. Lemma 2 implies that $k \leq \sqrt{n}$.

Let q be the task assigned to i at time t that caused i to become rich. Since $w_q \leq OPT$, $\sum_{j \in S} w_j \leq kOPT$ and $k \leq \sqrt{n}$ we conclude that

$$l_i(t) \leq \sqrt{n}L(t) + w_q + \sum_{j \in S} w_j \leq \sqrt{n}L(t) + OPT + \sqrt{n}OPT.$$

2.2.3 Tasks with Known Duration: the Assign1 Algorithm

This section considers the case where the duration of a task is known upon its arrival; *i.e.*, d_j is revealed to the on-line algorithm when task j arrives. We describe an algorithm [18, 20] whose competitive ratio is $O(\log nT)$, where T is the ratio of the maximum to minimum duration if the minimum possible task duration is known in advance. The algorithm is based on the on-line algorithm of [14], which solves the following "route allocation" problem: We are given a directed graph $G = (V, E)$ with $|V| = N$. Request i is specified as $(s_i, t_i, \{p_{i,e} | e \in E\})$, where $s_i, t_i \in V$ and for all $e \in E$, $\{p_{i,e}\} \geq 0$. Upon arrival of request i, the route allocation algorithm has to assign i to a path (route) P from s_i to t_i in G; the route assignments are permanent. Let $P = P_1, P_2, \ldots, P_k$ be the routes assigned to requests 1 through k by the on-line algorithm, and let $P^* = P_1^*, P_2^*, \ldots, P_k^*$ be the routes assigned by the off-line algorithm. Given a set of routes P, the load on edge e after the first j requests are satisfied is defined as:

$$l_e(j) = \sum_{i \leq j : e \in P_i} p_{i,e}.$$

Denote the maximum load as $\lambda(j) = \max_{e \in E} l_e(j)$. Similarly, define $l_e^*(j)$ and $\lambda^*(j)$ to be the corresponding quantities for the routes produced by the off-line algorithm. For simplicity, we will abbreviate $\lambda(k)$ as λ and $\lambda^*(k)$ as λ^*. The goal of the online algorithm is to produce a set of routes P that minimizes λ/λ^*. The online route allocation algorithm of [14] is $O(logN)$-competitive, where N is the number of vertices in the given graph. Roughly speaking, we will reduce our problem of the on-line load balancing of temporary tasks with known duration to several concurrent instances of the online route allocation for permanent routes problem, where $N = nT$. Then we will apply the algorithm of [14] to achieve an $O(logN) = O(lognT)$ competitive ratio . We assume that the minimum task duration is known in advance. Let $a(j)$ denote the arrival time of task j, and assume $a(1) = 0$. Let $T' = (\max_j a(j) + d(j))$ be the total duration of all the tasks. First, we make several simplifying assumptions and then show how to eliminate them:

- T' is known in advance to the on-line algorithm.
- T is known in advance to the on-line algorithm.
- Arrival times and task durations are integers; *i.e.*, time is discrete.

We now describe an $O(lognT')$-competitive algorithm called Assign1.

The Assign1 algorithm.

The idea is to translate each task into a request for allocating a route. We construct a directed graph G consisting of T' layers, each of which consists of

$n+2$ vertices, numbered $1, \ldots, n+2$. We denote vertex i in layer k as $v(i,k)$. For each layer $1 \leq k \leq T'$, and for $1 \leq i \leq n$, we refer to vertices $v(i,k)$ as common vertices. Similarly, for each layer k, $v(n+1,k)$ is referred to as the source vertex and $v(n+2,k)$ is referred to as the sink vertex. For each layer, there is an arc from the source $v(n+1,k)$ to each of the n common vertices $v(i,k+1)$. In addition, there is an arc from each common vertex $v(i,k)$ to the sink $v(n+2,k)$ in each layer. Finally, there is an arc from each common vertex $v(i,k)$ to the corresponding common vertex $v(i,k+1)$ in the next layer. The arc from $v(i,k)$ to $v(i,k+1)$will represent machine i during the time interval $[k,k+1)$ and the load on the arc will correspond to the load on machine i during this interval.

We convert each new task j arriving at $a(j)$ into a request for allocating a route in G from the source of layer $a(j)$ to the sink of layer $a(j)+d(j)$. The loads p_j are defined as follows: for arcs $v(i,k)$ to $v(i,k+1)$ for $a(j) \leq k \leq a(j)+d(j)-1$, we set $p_{i,e} = p_i(j)$; we set $p_{j,e}$ to 0 for the arcs out of the source of layer $a(j)$ and for the arcs into the sink of layer $a(j)+d(j)$, and to ∞ for all other arcs. Clearly, the only possible way to route task j is through the arcs $v(i,k)$ to $v(i,k+1)$ for $a(j) \leq k \leq a(j)+d(j)-1$ some i. This route raises the load on the participating arcs by precisely $p_i(j)$, which corresponds to assigning the task to machine i. Thus, minimizing the maximum load on an arc in the on-line route allocation on the directed graph G corresponds to minimizing the maximum machine load in the load balancing problem.

We now show how to construct an $O(logn T)$-competitive algorithm. Partition the tasks into groups according to their arrival times. Group m contains all tasks that arrive in the time interval $[(m-1)T, mT]$. Clearly, each task in group m must depart by time $(m+1)T$, and the overall duration of tasks in any group is at most $2T$. For each group, invoke a separate copy of Assign1 with $T' = 2T$. That is, assign tasks belonging to a certain group regardless of the assignments of tasks in other groups. Using the route allocation algorithm of [14], we get that for each group, the ratio between maximal on-line load to the maximal off-line load is at most $O(logn T)$. Moreover, at each instance of time, active tasks can belong to at most 2 groups. Thus, the maximal on-line load at any given time is at most twice the maximal on-line load of a single group. The off-line load is at least the largest load of the off-line algorithms over all groups, and hence the resulting algorithm is $O(logn T)$ -competitive. We now show how to remove the remaining simplifying assumptions. To remove the restriction that T is known in advance, let $T = d(1)$ when the first task arrives. If we are currently considering the m-th group of tasks, and task j arrives with $d(j) > T$, we set $T = d(j)$ and $T' = 2d(j)$. Observe that this does not violate the invariant that at any point in time the active tasks belong to at most two groups. We use the fact that the route allocation algorithm

of [14] is scalable, in the sense that the current assignment of tasks in group m is consistent with the assignment if Assign1 had used the new value of T' instead of the old one. Thus, the argument of $O(lognT)$-competitiveness holds as before. Finally, to eliminate the assumption that events coincide with the clock, that is, for all j, $a(j)$'s and $d(j)$'s are integral multiples of time units, Assign1 approximates $a(j)$ by $\lfloor a(j) \rfloor$ and $d(j)$ by $\lceil a(j) + d(j) \rceil - \lfloor a(j) \rfloor$. Since for all j, $d(j) \geq 1$, this approximation increases the maximal off-line load by at most a factor of 2. Thus, the following claim holds:

Theorem 2. *The above on-line load balancing algorithm with known task duration is $O(lognT)$-competitive.*

2.3 A Metaheuristic Approach

As can be inferred from the previous section, due to the on-line nature of the load balancing problem (as happens for the greater part of on-line problems), the approaches used to optimize the system are one-shot algorithms, *i.e.*, heuristics that incrementally build a solution over time; we recall that no reallocation is allowed. Nonetheless, we can reinterpret the RH algorithm as a sort of Stochastic hill-climbing method and, in particular, as a special class of Threshold Acceptance (TA) algorithm, which belongs to the class of metaheuristics.

The novelty of what we propose in the following stems also from the fact that the algorithm is still an on-line constructive algorithm, and thus guarantees reduced computing times, but acts as a more sophisticated approach, where a neighborhood search has to be made as for a local search method.

The striking difference between RH-like algorithms and metaheuristics lies in the fact that the former do not use an objective function to optimize explicitly, but use thresholds, e.g., see $L(t)$, to distinguish between "more profitable" solutions/choices and "less profitable" solutions/choices. This is why the RH algorithm can be associated with the TA algorithm.

The general framework in which a metaheuristic works is depicted in Table 2.1.

where $s^{(0)}$ is the initial solution and $s^{(t)}$ is the current solution at timestep t.

Stochastic hillclimbing methods escape from local minima by probabilistically accepting disimprovements, or "uphill moves". The first such method, Simulated Annealing (SA), was proposed independently by Kirkpatrick *et al.* [25] and Cerny [6] and is motivated by analogies between the solution space of an optimization instance and the microstates of a statistical thermodynamical ensemble. Table 2.2 summarizes the functionalities of the SA algorithm,

Table 2.1. Metaheuristic template

1. Let $s^{(0)}$ be an initial solution; $t = 0$.

2. Repeat until a stopping criterion is satisfied:

2.1. Find a local optimum s_l with local search starting from $s^{(t)}$.

2.2. Decide whether $s^{(t+1)} = s^{(t)}$ or $s^{(t+1)} = s^{(t-1)}$.

2.3. $t = t + 1$.

which uses the following criteria for Line 2.2 of Table 2.1. If s' is a candidate solution and the function f has to be minimized, and $f(s') < f(s_i)$, then $s_{i+1} = s'$, *i.e.*, the new solution is adopted. If $f(s') \geq f(s_i)$ the "hillclimbing" disimprovement to $s_{i+1} = s'$ still has a nonzero probability of being adopted, determined both by the magnitude of the disimprovement and the current value of a temperature parameter T_i. This probability is given by the "Boltzmann acceptance" criterion described in Line 3.3 of Table 2.2.

Table 2.2. SA template; max_it is a limit on number of iterations

1. Choose (random) initial solution s_0.

2. Choose initial temperature T_0.

3. For $i = 0$ to $max\_it - 1$

3.1. Choose random neighbor solution $s' \in N(s_i)$.

3.2. If $f(s') < f(s_i)$ then $s_{i+1} = s'$

3.3. else $s_{i+1} = s'$ with $Pr = exp((f(s_i) - f(s'))/T_i)$.

3.4. $T_{i+1} = next(T_i)$.

4. Return s_i, $0 \leq i \leq max\_it$, such that $f(s_i)$ is minimum.

In contrast to SA, TA relies on a threshold T_i, which defines the maximum disimprovement that is acceptable at the current iteration i. All disimprovements greater than T_i are rejected, while all improvements less than T_i are accepted. Thus, in contrast to the Boltzmann acceptance rule of annealing, TA offers a deterministic criterion as described in Line 2.2 of Table 2.1.

At timestep i, the SA temperature T_i allows hillclimbing by establishing a nonzero probability of accepting a disimprovement, while the TA threshold T_i allows hillclimbing by specifying a permissible amount of disimprovement. Typical SA uses a large initial temperature and a final temperature of zero (note that $T = \infty$ accepts all moves; $T = 0$ accepts only improving moves, *i.e.*, the algorithm behaves like a greedy algorithm). The monotone decrease in T_i is accomplished by $next(T_i)$, which is a heuristic function of the T_i value and

Table 2.3. TA template; *max_it* is a limit on the number of iterations.

1. Choose (random) initial solution s_0.
2. Choose initial threshold T_0.
3. For $i = 0$ to $max\_it - 1$
  3.1. Choose random neighbor solution $s' \in N(s_i)$.
  3.2. If $f(s') < f(s_i) + T_i$ then $s_{i+1} = s'$
    3.3. else $s_{i+1} = s_i$.
  3.4. $T_{i+1} = next(T_i)$.
4. Return s_i, $0 \leq i \leq max\_it$, such that $f(s_i)$ is minimum.

the number of iterations since the last cost function improvement (typically, $next(T_i)$ tries to achieve "thermodynamic equilibrium" at each temperature value). Similarly, implementations of TA begin with a large initial threshold T_0 which decreases monotonically to $T_i = 0$. Note that both SA and TA will in practice return the best solution found so far, *i.e.*, the minimum cost solution among $s_0, s_1, \ldots, s_{max\_it}$; this is reflected in Line 4 of Tables 2.2 and 2.3.

Going back to the analogy between the RH algorithm and Stochastic hill-climbing, we can in more detail associate the former algorithm with a particular class of TA algorithms denoted as Old Bachelor Acceptance (OBA) algorithms.

OBA uses a threshold criterion in Line 2.2 of Table 2.1, but the threshold changes dynamically – up or down – based on the perceived likelihood of it being near a local minimum. Observe that if the current solution s_i has lower cost than most of its neighbors, it will be hard to move to a neighboring solution; in such a situation, standard TA will repeatedly generate a trial solution s' and fail to accept it. OBA uses a principle of "dwindling expectations": after each failure, the criterion for "acceptability" is relaxed by slightly increasing the threshold T_i, see $incr(T_i)$ in Line 3.3 of Table 2.4 (this explains the name "Old Bachelor Acceptance"). After a sufficient number of consecutive failures, the threshold will become large enough for OBA to escape the current local minimum. The opposite of "dwindling expectations" is what we call ambition, whereby after each acceptance of s', the threshold is lowered (see $decr(T_i)$ in Line 3.2 of Table 2.4) so that OBA becomes more aggressive in moving toward a local minimum. The basic OBA is shown in Table 2.4.

Let us now examine what happens if we translate the RH algorithm in the realm of OBA (in Table 2.5 we show the RH algorithm modified in terms of OBA, which we have denoted as OBA-RH).

Table 2.4. OBA template; *max_it* is a limit on the number of iterations.

1. Choose (random) initial solution s_0.
2. Choose initial threshold T_0.
3. For $i = 0$ to $max\_it - 1$
  3.1. Choose random neighbor solution $s' \in N(s_i)$.
  3.2. If $f(s') < f(s_i) + T_i$ then $s_{i+1} = s'$ and $T_{i+1} = T_i - decr(T_i)$
    3.3. else $s_{i+1} = s_i$ and $T_{i+1} = T_i + incr(T_i)$.
4. Return s_i, $0 \leq i \leq max\_it$, such that $f(s_i)$ is minimum.

The first key-point to be addressed is how to take into account the objective function $f(s_i)$ that is not considered by the RH algorithm, and is an important issue in OBA.

Denote at timestep t the load l of machine i as $l_i^{(t)}$ and let $\mu(t)$ be the average load of the system at time t, *i.e.*, $\mu(t) = (\sum_i l_i^{(t-1)} + w_j)/n$. Moreover, let us define a candidate solution at timestep t as $s^{(t)}$. Thus, we can define

$$f(s^{(t)}) = \sum_i |l_i^{(t)} - \mu(t)|,$$

i.e., the sum of the absolute deviation from the average load of the system of each machine load. Similarly to how we have denoted a solution, let us denote the acceptance threshold at time t as $T^{(t)}$.

Following Line 1 of the algorithm in Table 2.4 we have to choose an initial solution. Solution $s^{(0)}$ in Line 1 of the OBA-RH algorithm is the empty solution, *i.e.*, the one in which all the machines are unloaded.

Without loss of generality, let us assume we are at timestep t, with solution $s^{(t-1)}$ of the previous step $(t-1)$. Following Line 3.1 of Table 2.4, we have to generate the set of neighboring solutions $N(s^{(t-1)})$ from $s^{(t-1)}$. Let us encode a solution in the neighborhood as follows: $s_1^{(t)}$ is the neighboring solution at timestep t that allocates incoming task j of weight w_j on machine 1, $s_2^{(t)}$, similarly, is the solution obtained by charging machine 2, and so on until the n-th solution in which machine n's load is increased by w_j.

Solution $s_i^{(t)}$ can be represented by vector $(l_1^{(t-1)}, \ldots, l_i^{(t-1)} + w_j, \ldots, l_n^{(t-1)})$, where the generic component k represents the load of machine k when j is assigned to such a machine; thus, it is clear that in $s_i^{(t)}$ all machine loads remain the same as in the previous iteration, except for machine i which is increased by w_j. Hence, the neighborhood of the problem is formed by n possible solutions, *i.e.*, the solutions obtainable by the current one adding, respectively, the new incoming task j to each of the n machines. In this case, choosing a

Table 2.5. OBA-RH template; *max_it* is a limit on the number of iterations

1. The initial solution $s^{(0)}$ is the one where all the machines are empty; set $T^{(0)} = \infty$.

2. For $t = 1$ to $max\_it - 1$

2.1. Let j be the task arriving at time t.

2.2. Evaluate the neighboring solutions $s_i^{(t)} \in N(s^{(t-1)})$.

2.3. If there are poor machines, then choose at random a neighboring solution s' corresponding to a poor machine; this will satisfy $f(s') < f(s^{(t-1)}) + T^{(t-1)}$; set $T^{(t)} \leq \max\{0, f(s') - f(s^{(t-1)})\}$;

2.4. else choose threshold $T^{(t)} \geq \max\{0, f(s_{\hat{i}}^{(t)}) - f(s^{(t-1)})\}$, where $\hat{i}$ is a rich machine whose windfall time is the smallest.

2.5. $s^{(t)} = s^{(t-1)}$.

3. Return $f(s^{(t)})$.

solution at random (see Line 3.1 of Table 2.4) means choosing a machine at random and then adding the incoming task to that machine.

For instance, at timestep $t = 1$, the solution $s^{(1)}$ can be chosen among the following n candidate solutions in $N(s^{(0)})$:

$$s_1^{(1)} = (w_j, 0, \ldots, 0),$$

$$s_2^{(1)} = (0, w_j, \ldots, 0),$$

$$\ldots$$

$$\ldots$$

$$s_n^{(1)} = (0, \ldots, 0, w_j).$$

Define a machine $\tilde{i}$ poor if $f(s_{\tilde{i}}^{(t)}) < f(s^{(t-1)}) + T^{(t-1)}$, and rich otherwise. Note that, since $f(s^{(0)})$ is 0 (all the machines are empty) and $\mu(0) = 0$, all the machines at time 1 will be rich if threshold $T^{(0)}$ is greater than $w_j + \frac{w_j}{n}(n-2)$. To let OBA-RH act as RH, we initially set $T^{(0)} = \infty$; in this way, all the machines are initially poor.

Thus, if one, or more than one, poor machine exists, a neighboring solution s' corresponding to a poor machine is chosen at random; according to the definition of a poor machine, this solution would also be accepted by the OBA algorithm. After solution acceptance, the threshold is decreased (see Line 2.3 of Table 2.5) by a quantity $decr(T^{(t)}) = T^{(t-1)} - f(s') + f(s^{(t-1)})$.

If in the neighborhood there is not a solution obeying Line 3.2 of Table 2.4, and, thus, in terms of OBA-RH a poor machine does not exist, we have to keep the previous solution; therefore, in the next iteration, the threshold is raised as done in Line 2.4 of Table 2.5 and the new solution is searched for among the same neighboring solutions (see Line 2.5 in Table 2.5).

In this case, it should appear clear why we have changed the notation, using t rather than i to indicate the timestep of the algorithm: in fact, when a solution is discarded, the algorithm cannot allocate the incoming task, since it has to modify the threshold in such a way that in the next iteration there is more chance of accepting a neighboring solution.

Thus, it could happen that at timestep t the number of tasks processed till t by the algorithm could be lower than t due to a possible rejection of allocation.

Setting the next threshold in the interval

$$f(s_{\hat{i}}^{(t)}) - f(s^{(t-1)}) \leq T^{(t)} \leq f(s_{\hat{\hat{i}}}^{(t)}) - f(s^{(t-1)})$$

where $\hat{\hat{i}}$ is the next rich machine after $\hat{i}$, allows one to select, in the next iteration, among the subset of poor machines that will be created having the same windfall time, and, in the case of a tie, the same minimum objective function increment; thus, if such a choice of the threshold is made, one can directly allocate task j to machine $\hat{i}$ without generating the successive neighborhood. In this case, we are doing exactly what RH does, by means of OBA. Note that to achieve such a condition, we can set the new threshold equal to

$$T^{(t)} = f(s_{\hat{i}}^{(t)}) - f(s^{(t-1)}) + \epsilon$$

where ϵ is a positive sufficiently small quantity, *i.e.*, $\epsilon \leq f(s_{\hat{\hat{i}}}^{(t)}) - f(s_{\hat{i}})$.

This version of OBA-RH, denoted OBA-RH revised, is depicted in Table 2.6.

Remark 1. Note that we have considered only the instants of time related to a task arrival, since a task departure is not a decision stage; it just decreases the machine load by a value w_j if j is the outcoming task.

2.4 Example

In the following, we provide an example of RH and OBA-RH to compare how they work.

Let us assume to have 4 machines and 5 incoming tasks. For the sake of simplicity, let us also assume that the departure dates of the tasks are

Table 2.6. OBA-RH revised template; *max_it* is a limit on the number of iterations

1. The initial solution $s^{(0)}$ is the one where all the machines are empty;
set $T^{(0)} = \infty$.
2. For $t = 1$ to $max\_it - 1$
2.1. Let j be the task arriving at time t.
2.2. Evaluate neighboring solutions $s_i^{(t)} \in N(s^{(t-1)})$.
2.3. If there are poor machines then choose at random a
neighboring solution s' corresponding to a poor machine;
this will satisfy $f(s') < f(s^{(t-1)}) + T^{(t-1)}$;
set $T^{(t)} = \max\{0, f(s') - f(s^{(t-1)})\}$;
2.4. else choose threshold $T^{(t)} = \max\{0, \min_i\{f(s_i^{(t)}) - f(s^{(t-1)})\}\} + \epsilon$,
where $\hat{i}$ is a rich machine whose windfall time is the smallest.
2.5. $s^{(t)} = s_{\hat{i}}^{(t)}$.
3. Return $f(s^{(t)})$.

all equal to 6 and that tasks arrive one by one at time 1, 2, 3, 4, and 5, respectively. Moreover, assume that the weights are as follows: $w_1 = 2, w_2 = 5, w_3 = 14, w_4 = 14, w_5 = 4$.

Assume task 1 is the first incoming task in the system; since $L(0) = 0$, we have that

$$L(1) = \max\{L(0), w_1, (w_1 + \sum_i l_i(0))/4\} = 2.$$

Thus, all the machines are poor, since their load is initially zero, and $\sqrt{n}L(1) = 4$.

We choose a poor machine at random, say machine 3; therefore, the current load vector is $p_1 = (0, 0, 2, 0)$. When task 2 arrives,

$$L(2) = \max\{L(1), w_2, (w_2 + \sum_i l_i(1))/4\} = 5.$$

Since $\sqrt{n}L(2) = 10$, all the machines are again poor and we can proceed by randomly choosing a machine, *e.g.*, machine 2. Our load vector is now $p_2 = (0, 5, 2, 0)$.

When task 3 enters the system, $L(3) = \max\{5, 14, 21/4\} = 14$, and again all the machines are poor since $\sqrt{n}L(3) = 28$. We then randomly assign task 3 to a machine, say machine 2. The new load vector is $p_3 = (0, 19, 2, 0)$.

In the next iteration, task 4 arrives and $L(4)$ is equal to 14. Again, it is easy to verify that all the machines are poor and let us suppose randomly choosing machine 2, whose load increases to 33.

Now, when task 5 arrives, $L(5) = 14$ and machine 2 is rich. Thus, we have to choose one poor machine, at random, among machines 1, 3 and 4. Let us suppose we choose machine 1 and the load vector p_5 is $(4, 33, 2, 0)$.

Let us now consider the OBA-RH algorithm. Since $T^{(1)} = +\infty$, all the machines are initially poor and thus we can allocate task 1 as we did for the RH algorithm, at random; as in the previous scenario, we choose machine 3. This corresponds to choosing a solution $s' = s_3^{(1)}$ in the neighborhood $N(s^{(0)})$ of $s^{(0)}$. Thus, the current load vector is the same as the load vector p_1 computed before.

Now, following Line 2.3 of Table 2.6, we set $T^{(2)} = \max\{0, f(s') - f(s^{(1)})\} = \max\{0, 3 - 0\} = 3$.

The next task to be considered is task 2. Evaluating the neighborhood of $s^{(1)}$ we obtain the following:

$$f(s_1^{(1)}) = (5 - 7/4) + 7/4 + (2 - 7/4) + 7/4 = 7,$$

$$f(s_2^{(1)}) = 7/4 + (5 - 7/4) + (2 - 7/4) + 7/4 = 7,$$

$$f(s_3^{(1)}) = 7/4 + 7/4 + (7 - 7/4) + 7/4 = 21/2,$$

$$f(s_4^{(1)}) = 7/4 + 7/4 + (2 - 7/4) + (5 - 7/4) = 7.$$

Thus, we have

$$f(s_1^{(1)}) - f(s^{(1)}) = 7 - 3 = 4,$$

$$f(s_2^{(1)}) - f(s^{(1)}) = 7 - 3 = 4,$$

$$f(s_3^{(1)}) - f(s^{(1)}) = 21/2 - 3 = 15/2,$$

$$f(s_4^{(1)}) - f(s^{(1)}) = 7 - 3 = 4.$$

It is easy to verify that all the machines are rich, since $f(s_i^{(1)}) - f(s^{(1)}) \geq T^{(2)}$ for each machine $i = 1, \ldots, 4$.

Thus, we cannot accept any solution, and have to increase the threshold and then repeat the neighborhood search for a new possibly acceptable solution.

The new threshold is

$$T^{(3)} = \max\{0, \min_i\{f(s_i^{(2)} - f(s^{(2)})\}\} + \epsilon = \max\{0, 7 - 3\} = 4 + \epsilon, \quad (2.1)$$

where, according to our definition of ϵ, we have that $\epsilon \leq f(s_{\hat{i}}^{(t)}) - f(s_{\hat{i}})$, *i.e.*, $\epsilon \leq 15/2 - 4 = 7/2$, and machines $\{1, 2, 4\}$ allow the achievement of $T^{(3)}$. Suppose we set $\epsilon = 7/2$.

Thus, we allocate task 2 to one of the rich machines in the set {1, 2, 4} since they all have the same minimum difference among $f(s_i^{(1)})$ and $f(s^{(1)})$.

We choose machine 2, set $s' = s_2^{(1)}$ and $s^{(2)} = s_2^{(1)}$, and the new load vector is $p_2 = (0, 5, 2, 0)$. Note that this choice of $T^{(3)}$ follows the same rationale as the one behind the OBA algorithm since we observe an increment in the threshold with respect to the previous iteration when a solution rejection occurs.

When task 3 arrives, we have the following situation:

$$f(s_1^{(2)}) = (14 - 21/4) + (21/4 - 5) + (21/4 - 2) + 21/4 = 35/2,$$

$$f(s_2^{(2)}) = 21/4 + (19 - 21/4) + (21/4 - 2) + 21/4 = 55/2,$$

$$f(s_3^{(2)}) = 21/4 + (21/4 - 5) + (16 - 21/4) + 21/4 = 43/2,$$

$$f(s_4^{(2)}) = 21/4 + (21/4 - 5) + (21/4 - 2) + (14 - 21/4) = 35/2.$$

Thus, we have

$$f(s_1^{(2)}) - f(s^{(2)}) = 35/2 - 7 = 21/2$$

$$f(s_2^{(2)}) - f(s^{(2)}) = 55/2 - 7 = 41/2,$$

$$f(s_3^{(2)}) - f(s^{(2)}) = 43/2 - 7 = 29/2,$$

$$f(s_4^{(2)}) - f(s^{(2)}) = 35/2 - 7 = 21/2.$$

It is easy to verify that, if in (2.1) we choose $\epsilon \leq 13/2$ then all the machines are rich because $f(s_i^{(2)}) - f(s^{(2)}) \geq T^{(3)}$ for each $i = 1, \ldots, 4$. Since we chose $\epsilon = 7/2$, we cannot accept any of the solutions in the neighborhood, and the next step is to increase the threshold to $T^{(4)} = 21/2 + \epsilon$ with $0 \leq \epsilon \leq 4$ and then accept one solution between $s_1^{(2)}$ and $s_4^{(2)}$. Suppose we select machine 1, *i.e.*, $s' = s_1^{(2)}, s^{(3)} = s'$, and the new load vector is $p_3 = (14, 5, 2, 0)$. Note that, also in this case, this choice of $T^{(4)}$ follows the rationale of the OBA algorithm where we observe an increment in the threshold with respect to the previous iteration when a solution rejection occurs.

When task 4 arrives, we have the following situation:

$$f(s_1^{(3)}) = (28 - 35/4) + (35/4 - 5) + (35/4 - 2) + 35/4 = 77/2,$$

$$f(s_2^{(3)}) = (14 - 35/4) + (19 - 35/4) + (35/4 - 2) + 35/4 = 31,$$

$$f(s_3^{(3)}) = (14 - 35/4) + (35/4 - 5) + (14 - 35/4) + 35/4 = 25,$$

$$f(s_4^{(3)}) = (14 - 35/4) + (35/4 - 5) + (35/4 - 2) + (14 - 35/4) = 21,$$

Thus, we have

$$f(s_1^{(3)}) - f(s^{(3)}) = 77/2 - 35/2 = 21,$$

$$f(s_2^{(3)}) - f(s^{(3)}) = 31 - 35/2 = 13,$$

$$f(s_3^{(3)}) - f(s^{(3)}) = 25 - 35/2 = 15/2,$$
$$f(s_4^{(3)}) - f(s^{(3)}) = 21 - 35/2 = 7/2.$$

It is easy to verify that machines 3 and 4 are poor, whatever the value of ϵ; while machine 2 is poor if $\epsilon \geq 5/2$, and machine 1 is rich regardless of ϵ. Assuming we have set $\epsilon = 0$, we have to choose one poor machine between machines 3 and 4. Suppose we choose machine 4, *i.e.*, $s^{(4)} = s_4^{(3)}$, our load vector is then $p_4 = (14, 5, 2, 14)$ and we set the threshold at

$$T^{(3)} = \max\{0, \{f(s_4^{(3)} - f(s^{(3)})\}\} = 7/2.$$

Note that, as with the OBA algorithm, the threshold is decreased due to a solution acceptance.

When task 5 arrives, we have the following situation:

$$f(s_1^{(4)}) = (18 - 35/4) + (35/4 - 5) + (35/4 - 2) + (14 - 35/4) = 25,$$
$$f(s_2^{(4)}) = (14 - 35/4) + (9 - 35/4) + (35/4 - 2) + (14 - 35/4) = 175/4,$$
$$f(s_3^{(4)}) = (14 - 35/4) + (35/4 - 5) + (35/4 - 6) + (14 - 35/4) = 17,$$
$$f(s_4^{(4)}) = (14 - 35/4) + (35/4 - 5) + (35/4 - 2) + (18 - 35/4) = 25,$$

Thus, we have

$$f(s_1^{(4)}) - f(s^{(4)}) = 25 - 21 = 4,$$
$$f(s_2^{(4)}) - f(s^{(4)}) = 175/4 - 21 = 91/4,$$
$$f(s_3^{(4)}) - f(s^{(4)}) = 17 - 21 = -4,$$
$$f(s_4^{(4)}) - f(s^{(4)}) = 25 - 21 = 4.$$

It is easy to verify that only machine 3 is poor, and thus, we allocate task 5 to this machine to obtain a final load vector equal to $p_5 = (14, 5, 7, 14)$.

Note that the vector so obtained is the best possible solution obtainable by an off-line algorithm; the overall unbalance of the system at the end of the allocation is equal to

$$(14 - 10) + (10 - 5) + (10 - 7) + (14 - 10) = 16,$$

being that $\mu(5) = 19$.

We conclude noting that, based on the hypothesis made at the beginning of the example, after the arrival of task 5, the system is emptied since all the tasks leave the machines.

2.5 Experimental Results

We have experimented the greedy , the semi-greedy, the RH, and the OBA-RH revised algorithms on random instances with:

- 100, 150, 200, 250, 300, 350, 400, 450 and 500 tasks,
- a number of machines from 5 to 20,
- weights $w_j \in \{1, \ldots, 10\}$ assigned at random to each task j,
- arrival date of the tasks is chosen at random in the time interval $[1, 360]$,
- the duration of a task varies at random from 1 to 10 time units.

All the algorithms were implemented in the C language and run on a PC with 2.8 MHz Intel Pentium processor and 512 MB RAM. In this set of experiments the input to the algorithms is a list of events, *i.e.*, an ordered sequence of arrival dates and departure dates of the tasks, that cannot be exploited in advance by the algorithms due to the on-line nature of the problem. Hence, starting from the first event (that must be an arrival time of a certain task since we assume the system empty), the algorithms in case of an incoming task decide the machine onto which allocate such a task, and simply update the load of a machine when the event to be processed is the departure of a task.

The objective function used to obtain the values given in Tables 2.7-2.10 is $\sum_i |\mu(t) - l_i(t)|$. These are the values produced by the algorithm at the end of its run. Note that the results are given as the superior integer part of the objective function. Moreover, note that the results obtained for the semi-greedy algorithm are achieved by fixing $r = \lceil 0.2 \cdot$ number of tasks $\rceil$.

Table 2.7. Comparison among different on-line load balancing algorithms. The number of machines is 5

	Greedy	Semi-greedy	RH	OBA-RH revised
# tasks: 100	112	108	95	89
150	148	138	122	102
200	201	189	175	140
250	244	225	202	182
300	302	295	255	221
350	341	312	277	256
400	389	365	299	268
450	412	378	325	272
500	478	452	332	289

Table 2.8. Comparison among different on-line load balancing algorithms. The number of machines is 10

	Greedy	Semi-greedy	RH	OBA-RH revised
# tasks: 100	60	54	47	43
150	74	69	61	51
200	100	99	87	120
250	125	112	101	91
300	154	158	128	110
350	178	160	135	128
400	195	182	150	130
450	202	195	161	135
500	235	225	165	147

Table 2.9. Comparison among different on-line load balancing algorithms. The number of machines is 15

	Greedy	Semi-greedy	RH	OBA-RH revised
# tasks: 100	43	35	33	27
150	44	43	41	31
200	50	48	47	44
250	68	65	61	60
300	87	80	102	74
350	98	90	111	77
400	112	102	115	79
450	150	142	122	85
500	180	178	125	92

Results in the tables highlight the behavior of the four algorithms. The greedy algorithm has the worst performance once all the tasks have been processed. This is not surprising since, as we mentioned in the previous section, the greedy algorithm is able to do better in the first iterations of the algorithm run, while it tends to jeopardize solutions over time. To emphasize this, in Table 2.1 we show the trend of the objective function values over time. It should be noted how the greedy algorithm is able to maintain a good (low) objective function value for the first iterations, while this value grows quickly and is not able to consistently reduce the objective function values.

Table 2.10. Comparison among different on-line load balancing algorithms. The number of machines is 20

	Greedy	Semi-greedy	RH	OBA-RH revised
# tasks: 100	35	31	28	21
150	38	34	29	25
200	50	45	32	27
250	52	44	38	34
300	55	50	47	40
350	62	58	48	44
400	68	62	59	51
450	70	72	59	57
500	85	75	59	68

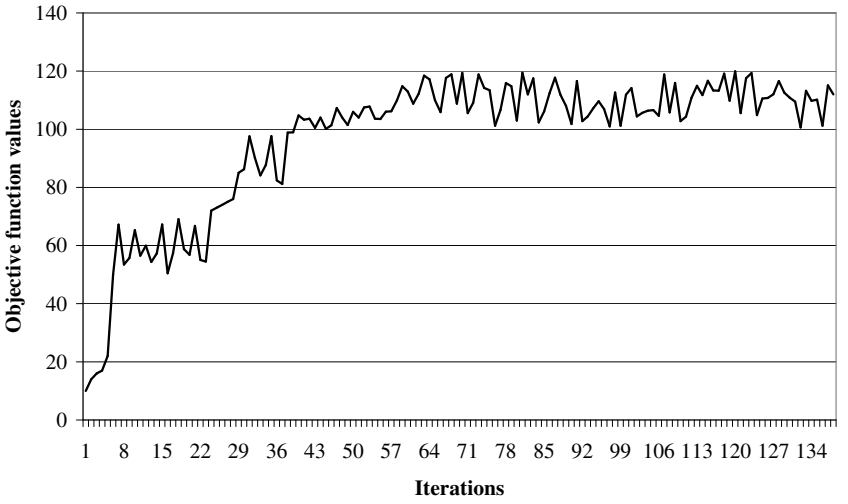

Fig. 2.1. The trend of the objective function of the greedy algorithm over time: the instance with 100 tasks and 5 machines

The semi-greedy algorithm is able to perform better than the greedy algorithm. For the sake of completeness, in Figure 2.2 we show the trend of the objective function values obtained at the end of the algorithm run, varying the values of r, *i.e.*, the cardinality of the restricted candidate list, for the case of 100 tasks and 5 resources. As can be seen, the best values are obtained in correspondence to 0.2, and this justifies our choice of r.

When using the RH and the OBA-RH revised algorithms, we observe a further decrease in the objective function value in the last stage of the

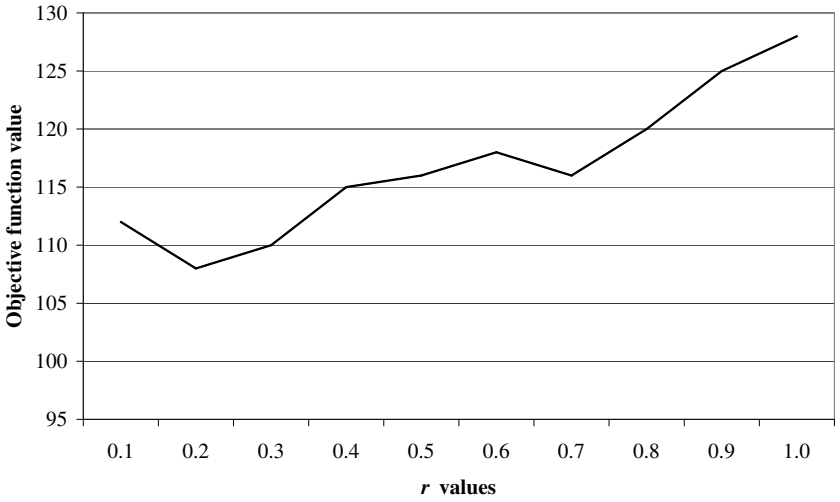

Fig. 2.2. The trend of the objective function of the semi-greedy algorithm

algorithm; in particular, we see that the latter algorithm beats the former, producing results that are up to about 40% better.

To allow a fair comparison, in Tables 2.11-2.14 we showed the average objective function values over the number of iterations. We observe that the general behavior does not change, while the performance gap among the algorithms is enforced.

Table 2.11. Comparison among different on-line load balancing algorithms. The average case with 5 machines

	Greedy	Semi-greedy	RH	OBA-RH revised
# tasks: 100	89.6	86.4	76.0	71.2
150	118.4	110.4	97.6	81.6
200	160.8	151.2	140.0	112.0
250	195.2	180.0	161.6	145.6
300	241.6	236.0	204.0	176.8
350	272.8	249.6	221.6	204.8
400	311.2	292.0	239.2	214.4
450	329.6	302.4	260.0	217.6
500	382.4	361.6	265.6	231.2

Table 2.12. Comparison among different on-line load balancing algorithms. The average case with 10 machines

	Greedy	Semi-greedy	RH	OBA-RH revised
# tasks: 100	49.6	46.4	46.0	41.2
150	72.4	60.4	58.6	41.6
200	80.8	78.2	70.0	65.0
250	92.2	86.0	81.6	75.6
300	120.6	112.0	102.0	89.8
350	135.8	124.6	118.6	104.8
400	165.2	144.0	125.2	114.4
450	178.6	165.4	136.0	117.6
500	200.4	189.6	150.6	131.2

Table 2.13. Comparison among different on-line load balancing algorithms. The average case with 15 machines

	Greedy	Semi-greedy	RH	OBA-RH revised
# tasks: 100	39.6	36.4	36.0	31.2
150	48.4	42.4	37.6	31.6
200	54.8	51.2	46.0	42.0
250	65.2	62.0	61.6	55.6
300	71.6	66.0	70.0	66.8
350	82.8	79.6	71.6	70.8
400	91.2	92.0	89.2	84.4
450	109.6	102.4	100.0	97.6
500	122.4	121.6	115.6	111.2

In Figures 2.3-2.6, we showed the shape of the load of the machines for instances with 200 and 500 tasks, respectively, and 10 machines, for the greedy and the OBA-RH revised algorithms.

2.5.1 A Multi-objective Approach in the Case of Known Task Departure Dates

As for the case of known durations, we can improve the OBA-RH revised algorithm, by considering a new multi-objective function.

In fact, contrary to the previous case, when the duration of a task is known upon its arrival, we can adopt the following objective function:

Table 2.14. Comparison among different on-line load balancing algorithms. The average case with 20 machines

	Greedy	Semi-greedy	RH	OBA-RH revised
# tasks: 100	29.6	26.4	21.0	19.2
150	38.4	35.4	28.6	25.6
200	40.8	38.2	35.0	32.0
250	52.2	46.0	37.6	34.6
300	60.6	52.0	40.0	43.8
350	65.8	56.6	44.6	55.8
400	75.2	65.0	48.2	70.4
450	87.6	77.4	65.0	72.6
500	98.4	85.6	80.6	75.2

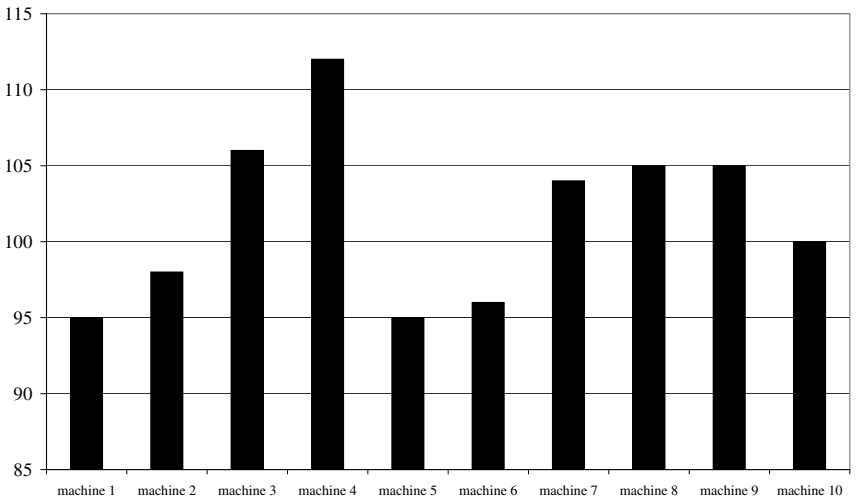

Fig. 2.3. Schedule shape produced by the greedy algorithm on an instance with 200 tasks and 10 machines

$$\min_i \alpha \cdot M_i(t) + \beta \cdot \bar{M}_i(t+1, \Delta t)$$

where:

- $M_i(t)$ is the load of machine i once the incoming task is associated with i.
- $\bar{M}_i(t+1, \Delta t)$ is the the average load of machine i in the interval $[t+1, \Delta t]$.
- α and β are two parameters in $[0, 1]$ whose sum is 1.

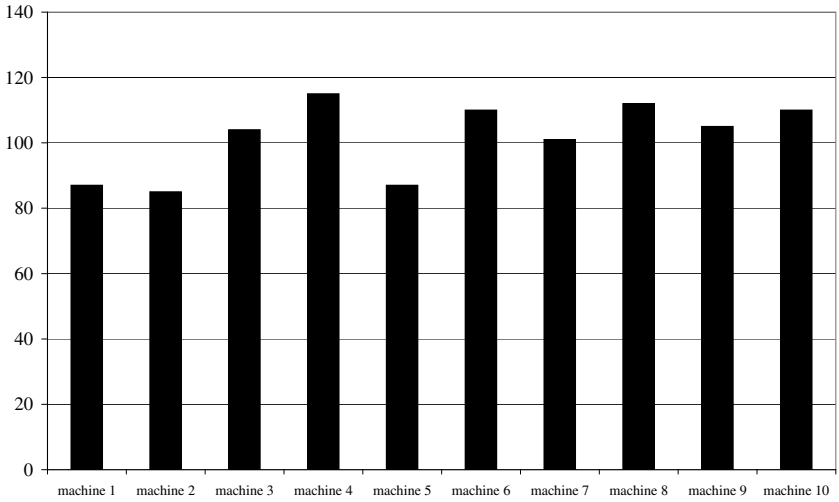

Fig. 2.4. Schedule shape produced by the OBA-RH revised algorithm on an instance with 200 tasks and 10 machines

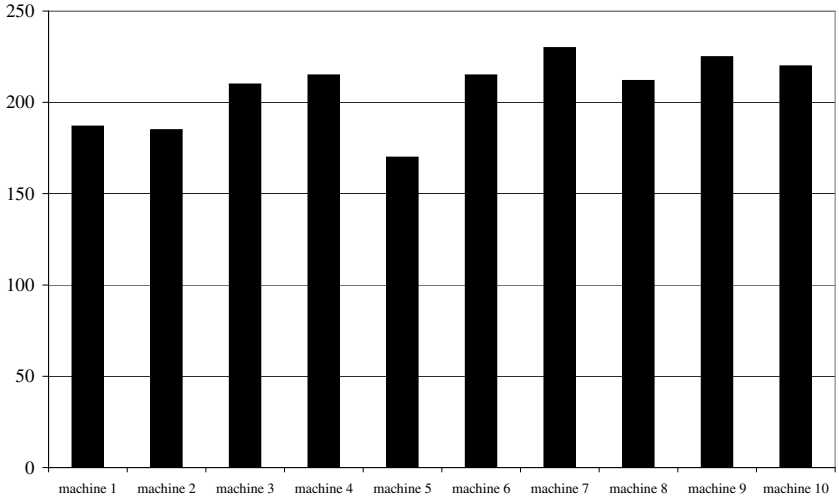

Fig. 2.5. Schedule shape produced by the greedy algorithm on an instance with 500 tasks and 10 machines

Note that in case $\beta = 0$ the objective function reduces to minimize the maximum load.

Example 1. Suppose that the system has three machines and that at time t the incoming task j has weight $w_j = 2$ with duration equal to $d_j = 2$; moreover, assume that assigning task j to machine 1 produces the situation in Figure

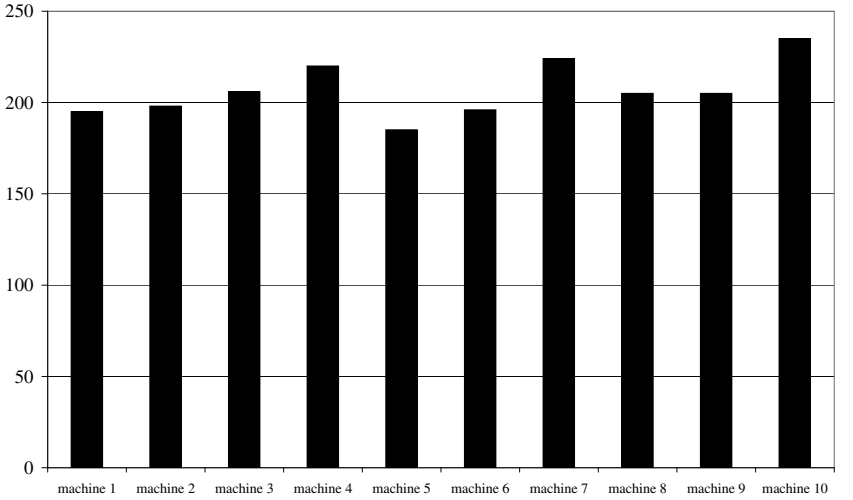

Fig. 2.6. Schedule shape produced by the OBA-RH revised algorithm on an instance with 500 tasks and 10 machines

2.7, and assigning task j to machine 2 produces the situation in Figure 2.8, assigning task j to machine 3 produces the situation in Figure 2.9.

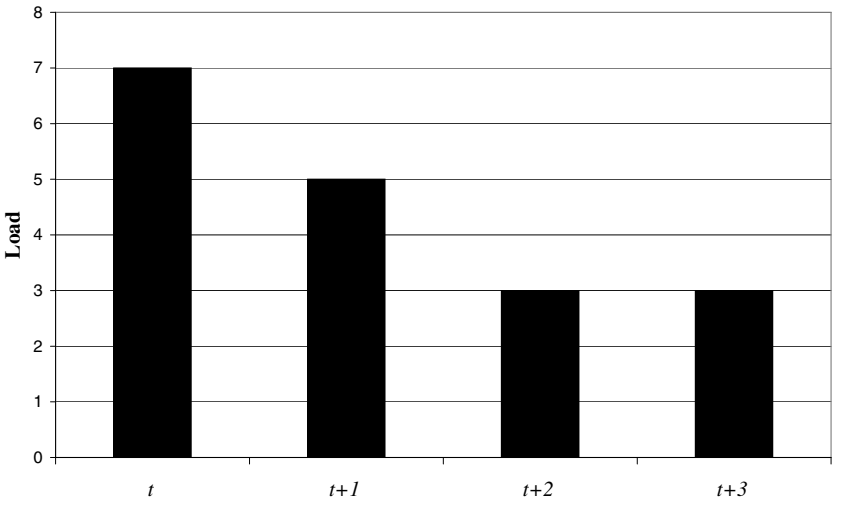

Fig. 2.7. Load shape of machine 1 over time interval $[t, t+3]$

It is easy to see that if $\Delta t = 2$:

$$\alpha \cdot 7 + \beta \cdot 4$$

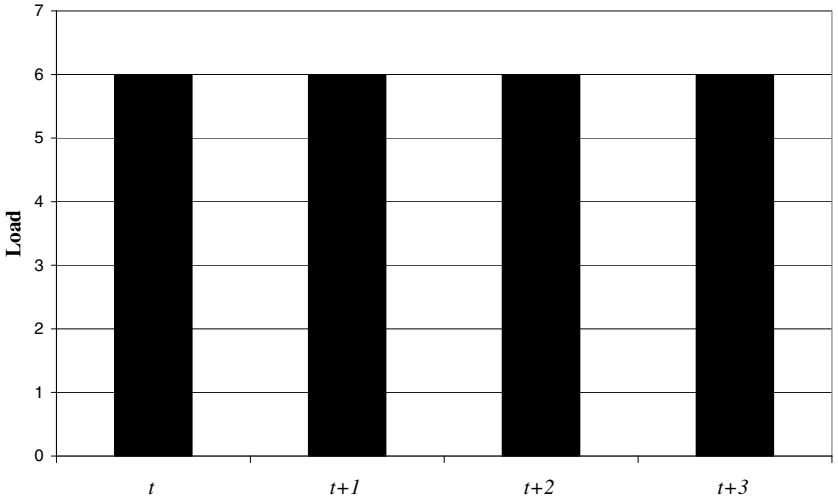

Fig. 2.8. Load shape of machine 2 over time interval $[t, t+3]$

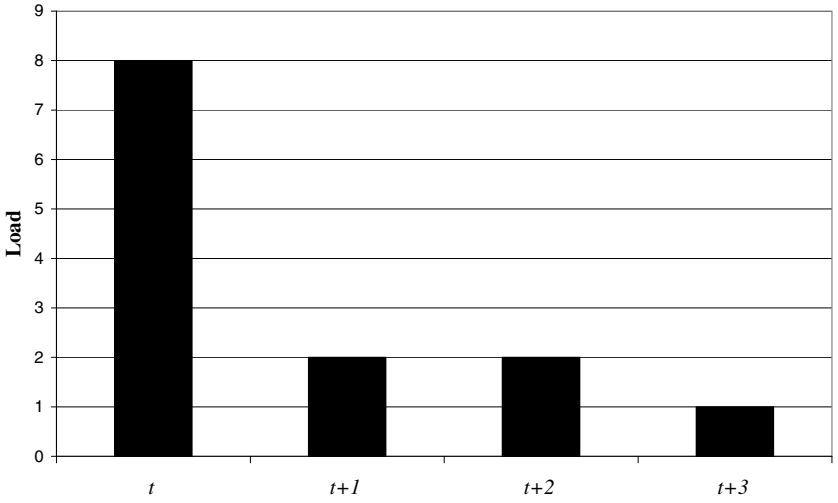

Fig. 2.9. Load shape of machine 3 over time interval $[t, t+3]$

$$\alpha \cdot 6 + \beta \cdot 6$$

$$\alpha \cdot 8 + \beta \cdot 2.$$

Based on the values of α and β we have a different machine choice. For instance if we have $\beta = 0.8$, we have that the objective function is $\min\{4.6, 6, 3.2\} = 3.2$ and the choice is that of machine 3. On the contrary, if $\beta = 0.2$ then the objective function is $\min\{6.4, 6, 7.6\} = 6$, and the choice is machine 2.

In Tables 2.15-2.18 we compare the results obtained by OBA-RH revised implemented with the multi-objective function (denoted as OBA-RH$_{rm}$), and the greedy, semi-greedy, and RH algorithms, with $\alpha = 0.7$, $\beta = 0.3$ and $\Delta t = 3$. The latter values of the parameters α, β and Δ are those that gave on average the better results for all the algorithms as suggested by the tuning reported in Figure 2.10 for the case of 100 tasks and 10 machines.

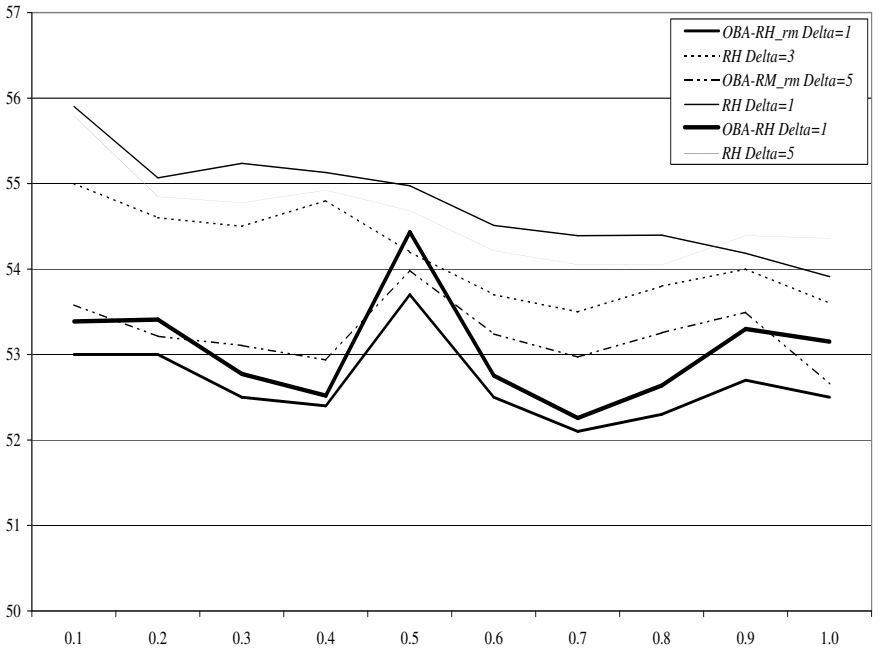

Fig. 2.10. Tuning of the multi-objective parameters

Results on the multi-objective scenario show that OBA-RH$_{rm}$ is able, as in the single-objective case, to perform better than the competing algorithms.

2.6 Conclusions

In this chapter, we have studied a well known problem in production processes: the balancing of the work load over a set of facilities. This problem is highly dynamic, that is, in practice, the assignment of an incoming task to a machine must be done *during* process execution without any knowledge of future tasks. We proposed and compared four algorithms. Starting from the simplest greedy heuristic algorithm to a more structured meta-heuristic

Table 2.15. Comparison among different on-line load balancing algorithms with a multi-objective approach. The average case with 5 machines

	Greedy	Semi-greedy	RH	OBA-RH$_{rm}$
# tasks: 100	121.5	120.7	118.2	117.1
150	178.4	176.5	173.4	169.6
200	238.6	236.3	233.6	226.9
250	296.8	293.2	288.8	284.9
300	358.0	356.6	349.0	342.4
350	415.5	409.9	403.2	399.2
400	474.7	470.1	457.4	451.5
450	529.1	522.6	512.4	502.2
500	591.8	586.8	563.7	555.5

Table 2.16. Comparison among different on-line load balancing algorithms with a multi-objective approach. The average case with 10 machines

	Greedy	Semi-greedy	RH	OBA-RH$_{rm}$
# tasks:100	57.7	56.5	55.5	52.1
150	84.4	82.5	81.4	77.1
200	110.1	108.9	107.3	103.3
250	135.9	134.5	133.1	128.8
300	163.0	161.4	159.7	154.5
350	189.2	187.3	185.9	180.2
400	216.4	213.8	211.4	205.7
450	242.4	240.4	238.2	230.9
500	269.0	267.2	263.3	256.6

approach. Improvements in solution quality were very significant in all instances, achieving, for example, almost 40% when comparing the OBA-RH revised algorithm solution to the one produced by the greedy procedure for the instance of 500 tasks and 5 machines.

Table 2.17. Comparison among different on-line load balancing algorithms with a multi-objective approach. The average case with 15 machines

	Greedy	Semi-greedy	RH	OBA-RH$_{rm}$
# tasks: 100	40.3	39.1	38.0	34.9
150	57.6	56.2	54.8	51.6
200	74.8	73.5	72.1	68.8
250	92.2	91.0	90.0	86.1
300	109.4	108.0	107.3	103.3
350	126.9	125.6	124.0	120.2
400	144.2	143.2	142.0	137.6
450	162.2	160.7	159.5	154.9
500	179.8	178.8	177.3	172.2

Table 2.18. Comparison among different on-line load balancing algorithms with a multi-objective approach. The average case with 20 machines

	Greedy	Semi-greedy	RH	OBA-RH$_{rm}$
# tasks: 100	27.2	27.0	26.6	26.0
150	44.4	43.2	41.6	38.8
200	57.1	55.9	54.6	51.6
250	70.4	69.0	67.3	64.2
300	83.5	81.9	80.0	77.2
350	96.4	94.7	92.8	90.3
400	109.6	107.9	107.6	103.5
450	123.1	121.3	119.4	116.1
500	136.4	134.4	133.0	128.8