Online Load Balancing Made Simple: Greedy Strikes Back*

Pilu Crescenzi[1], Giorgio Gambosi[2], Gaia Nicosia[3], Paolo Penna[4]**, and Walter Unger[5]

[1] Dipartimento di Sistemi ed Informatica, Università di Firenze, via C. Lombroso 6/17, I-50134 Firenze, Italy (piluc@dsi.unifi.it)
[2] Dipartimento di Matematica, Università di Roma "Tor Vergata", via della Ricerca Scientifica, I-00133 Roma, Italy (gambosi@mat.uniroma2.it)
[3] Dipartimento di Informatica e Automazione, Università degli studi "Roma Tre", via della Vasca Navale 79, I-00146 Roma, Italy (nicosia@dia.uniroma3.it)
[4] Dipartimento di Informatica ed Applicazioni "R.M. Capocelli", Università di Salerno, via S. Allende 2, I-84081 Baronissi (SA), Italy (penna@dia.unisa.it),
[5] RWTH Aachen, Ahornstrasse 55, 52056 Aachen, Germany (quax@cs.rwth-aachen.de)

Abstract. We provide a new simpler approach to the on-line load balancing problem in the case of restricted assignment of temporary weighted tasks. The approach is very general and allows to derive on-line distributed algorithms whose competitive ratio is characterized by some combinatorial properties of the underlying graph representing the problem.

The effectiveness of our approach is shown by the hierarchical server model introduced by Bar-Noy *et al* '99. In this case, our method yields simpler and distributed algorithms whose competitive ratio is at least as good as the existing ones. Moreover, the resulting algorithms and their analysis turn out to be simpler. Finally, in all cases the algorithms are optimal up to a constant factor.

Some of our results are obtained via a combinatorial characterization of those graphs for which our technique yields $O(\sqrt{n})$-competitive algorithms.

1 Introduction

Load balancing is a fundamental problem which has been extensively studied in the literature because of its many applications in resource allocation, processor scheduling, routing, network communication, and many others. The problem is to assign tasks to a set of n processors, where each task has an associated

* A similar title is used in [13] for a facility location problem.

** supported by the European Project IST-2001-33135, Critical Resource Sharing for Cooperation in Complex Systems (CRESCCO). Work partially done while at the Dipartimento di Matematica, Università di Roma "Tor Vergata" and while at the Institut für Theoretische Informatik, ETH Zentrum.

J.C.M. Baeten et al. (Eds.): ICALP 2003, LNCS 2719, pp. 1108–1122, 2003.

load vector and duration. Tasks must be assigned immediately to exactly one processor, thereby increasing the load of that processor by the amount specified by the corresponding coordinate of the load vector for the duration of the task. Usually, the goal is to minimize the maximum load over all processors.

The *on-line* load balancing problem has several natural applications. For instance, consider the case in which processors represent channels and tasks are communication requests which arrive one by one. When a request is assigned to a channel, a certain amount of its bandwidth is reserved for the duration of the communication. Since channels have limited bandwidth, the maximum load is an important measure here.

Several variants have been proposed depending on the structure of the load vectors, whether we allow *preemption* (i.e., to reassign tasks), whether tasks remain in the system "forever" or not (i.e., permanent vs. temporary tasks), whether the (maximum) duration of the tasks is known, and so on [1,3,4,5,6,14, 16] (see also [2] for a survey).

In this paper, we study the on-line load balancing problem in the case of *temporary tasks* with *restricted assignment* and *no preemption*, that is:

- Tasks arrive one by one and their duration is unknown.
- Each task can be assigned to one processor among a subset depending on the type of the task.
- Once a task has been assigned to a processor, it cannot be reassigned to another one.
- Assigning a task to a processor increases the corresponding load by an amount equal to the weight of the task.

The problem asks to find an assignment of the tasks to the processors which minimizes the maximum load over all processors and over time.

Among others, this variant has a very important application in the context of wireless networks. In particular, consider the case in which we are given a set of base stations and each mobile user can only connect to a subset of them: those that are "close enough". A user may unexpectedly appear in some spot and ask for a connection at a certain transmission rate (i.e., bandwidth). Also, the duration of this transmission is not specified (this is the typical case of a telephone call). Because of the application, it is desirable not to reassign users to other base stations (i.e., to avoid the handover), unless this becomes unavoidable because a user moves away from the transmission range of its current base (in the latter case, we can model this as a termination of the current request and a new request appearing in the new position).

As usual, we compare the cost of a solution computed by an on-line algorithm with the best off-line algorithm which minimizes the maximum load *knowing the entire sequence* of task arrivals and departures. Informally, an on-line algorithm is r-competitive if, at any instant, its maximum processor load is at most r times the optimal maximum processor load.

It is convenient to formulate our problem by means of a bipartite graph with vertices corresponding to processors and possible "task types". More formally, let $P = \{p_1, \ldots, p_n\}$ be a set of processors and let $\mathcal{T} \subseteq 2^P$ be a set of *task types*.

We represent the set of task types by means of an *associated bipartite graph* $G_{P,\mathcal{T}}(X_{\mathcal{T}} \cup P, E_{\mathcal{T}})$, where $X_{\mathcal{T}} = \{x_1, \ldots, x_{|\mathcal{T}|}\}$ and

$$E_{\mathcal{T}} = \{(x_i, p_j) \mid p_j \text{ belongs to the } i\text{-th element of } \mathcal{T}\}.$$

A *task* t is a pair (x, w), where $x \in X_{\mathcal{T}}$ and w is the positive integer weight of t. The set of processors to which t can be *assigned* is $P_t = \{p| \ (x, p) \in E_{\mathcal{T}}\}$, that is, the set of nodes of $G_{P,\mathcal{T}}(X_{\mathcal{T}} \cup P, E_{\mathcal{T}})$ that are adjacent to x. In our example of mobile users above, the type of a task corresponds to the user's position[1]. In general, we consider two tasks which can be assigned to the same set of processors as belonging to the same type.

We follow the intuition that "nice" graphs may yield better competitive ratios, as one can see with the following examples:

General case. We do not assume anything regarding the possible task types. So, the graph must contain all possible task types corresponding to any subset of processors, that is, $\mathcal{T} = 2^P$. Under this assumption, the best ratio achievable is $\Theta(\sqrt{n})$ [3,5] (see also [17]), while the greedy algorithm is exactly $\frac{3n^{2/3}}{2}(1 + o(1))$-competitive [3], thus not optimal.

Identical machines. There is only one task type since a task can be assigned to any of the machines. Therefore, the graph is the complete bipartite graph $K_{1,n}$ and the competitive ratio of the problem is $\Theta(2 - 1/n)$. This ratio is achieved by the greedy algorithm [11,12], which is optimal [4].

Hierarchical servers. Processors are totally ordered and the type of a task corresponds to the "rightmost" processor that can execute that task. The set $\mathcal{T}$ contains one node per processor and the i-th node of $\mathcal{T}$ is adjacent to all processors j, with $1 \leq j \leq i$. There exists a 5-competitive algorithm and the greedy algorithm is at least $\Omega(\log n)$-competitive.

Noticeably, one can consider the first two cases as the two extremes of our problem, because of both the (non-) optimality of the greedy algorithm and the (non-) constant competitive ratio of the optimal on-line algorithm. From this point of view, the latter problem is somewhat in between.

A related question is whether the greedy approach performs badly because of the fact that it must decide where to assign a task only based on *local* information (i.e., the current load of those processors that can execute that task). Indeed, the optimal algorithm in [5, Robin-Hood Algorithm] requires the computation of (an estimation of) the off-line optimum, which seems hard to compute in this local fashion. The algorithms in [7], too, require the computation of a quantity related to the optimum which depends on the current assignment of tasks of several types (see [7, Algorithm Continuous and Optimal Lemma]).

The idea of exploiting combinatorial properties of the graph $G_{P,\mathcal{T}}(X_{\mathcal{T}} \cup P, E_{\mathcal{T}})$ has been first used in [9]. In particular, the approach in [9] is based on the construction of a suitable subgraph which is used by the greedy algorithm (in

[1] Clearly, this is a simplification of the reality where other constraints must also be taken into account.

place of the original one). This subgraph is the union of a set of complete bipartite subgraphs (called clusters). So, this method can be seen as a modification of the greedy algorithm where the topology of the network is taken into account in order to limit the possible choices of the greedy algorithm[2] . Therefore, the resulting algorithms only use "local information" as the greedy one does.

Several topologies have been considered in [9] for which the method improves over the greedy algorithm and matches the lower bound of the problem(s). In all such cases, however, the improvement is only by a constant factor, since the greedy algorithm was already $O(1)$-competitive.

The main contribution of this paper (see Sect. 2) is a new approach to the problem based on the construction of a suitable subgraph to be used by the greedy algorithm. In this sense, our work is similar in spirit to [9]. However, the results here greatly improve over the method in that paper. Indeed, we show that:

- Some problems cannot be optimally solved with the solution in [9], while our approach does yield optimal competitive ratios.
- Our approach subsumes the one in [9] since the latter can be seen as a special case of the one presented here.

Also, our method yields the first example in which there is a significant improvement w.r.t. the greedy algorithm. This arises from the relevant case of hierarchical topologies, for which we attain a competitive ratio of 5 (4 for unweighted tasks[3]), while the greedy algorithm is at least $\Omega(\log n)$ in both cases. Table 1 summarizes the results obtained for these topologies. Even though, when $n \to \infty$, we achieve the same competitive ratio of [7], our algorithms and their analysis turn out to be much simpler. (Actually, for fixed n, our analysis yields strictly better ratios.)

We then turn our attention to the general case. In general, it might be desirable to automatically compute the best subgraph, as this would also give a simple way to test the goodness of our method w.r.t. a given graph. Unfortunately, one of our results is the NP-hardness of the problem of computing an optimal or even a c-approximate solution, for some constant $c > 1$.

In spite of this negative result, we demonstrate that a "sufficiently good" subgraph can be obtained very easily in many cases. We first provide a *sufficient condition* for obtaining $O(\sqrt{n})$-competitive algorithms with our technique: the existence of a b-matching in (a suitable subgraph of) the graph $G_{\mathcal{P},\mathcal{T}}$, for some constant b independent of n. Notice that, the lower bound for the general case is $\Omega(\sqrt{n})$, which applies to randomized algorithms [3] and to sequences of tasks of length polynomial in n [17]. By using this result, we obtain a $(2\sqrt{n}+2)$-

[2] This approach is somewhat counterintuitive since the algorithm improves when adding further restrictions to it (not to the adversary). This is reminiscent of the well-known Braess' paradox [8,15], where the removal of some edges from a graph unexpectedly improves the latency of the flow at Nash equilibrium.

[3] We denote the version of these problems in which all tasks have weight one by *unweighted*.

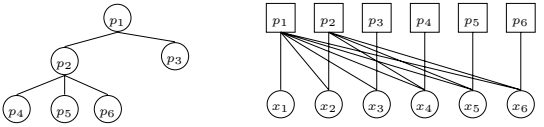

Fig. 1. An example of tree hierarchy (left) and the corresponding bipartite graph (right).

competitive distributed algorithm for the hierarchical server version in which processors are ordered as in a rooted tree (see the example in Fig. 1). A $\Omega(\sqrt{n})$ lower bound for centralized on-line algorithms also applies to this restriction [7], thus implying the optimality of our result. Additionally, we can achieve the same upper bound also when the ordering of the processors is given by *any* directed graph. This bound is only slightly worse than the $2\sqrt{n}+1$ given by the Robin-Hood algorithm in [5]. All algorithms obtained with our technique can be

Table 1. Performance of our method in the case of hierarchical servers.

	Our Method	Previous Best	Greedy
Weighted	$5n/(n+4)$ [Th. 5]	5 [7]	$\Omega(\log n)$ [folklore]
Unweighted	$4n/(n+4)$ [Th. 5]	4 [7]	$\Omega(\log n)$ [folklore]

considered distributed in that they compute a task assignment only based on the current load of those processors that can be used for that task. This is a rather appealing feature since, in applications like the mobile networks above, introducing a global communication among bases for every new request to be processed may turn out to be unfeasible. Additionally, in several cases considered here (linear and tree hierarchical topologies), the construction of the subgraph is used solely for the analysis, while the actual on-line algorithm does not require any pre-computation (although the algorithm is different from the greedy one and it implements the subgraph used in the analysis).

Finally, we believe that our analysis is per se interesting since it translates the notion of adversary into a combinatorial property of the subgraph we are able to construct during an off-line preprocessing phase. As a by-product, the analysis of our algorithms is simpler and more intuitive.

Roadmap. We introduce some notation and definitions in Sect. 1.1. The technique and its analysis are presented in Sect. 2. The hardness results are given in Sect. 2.2. We give a first application to the hierarchical topologies in Sect. 3. The application to the general case is described in Sect. 4 where we provide sufficient conditions for $O(\sqrt{n})$-competitiveness. These results are used in Sect. 4.1 where we obtain the results on generalized server hierarchies. Finally, in Sect. 5 we discuss some further features of our algorithms and present some open problems.

Due to lack of space, some of the proofs are only sketched or omitted in this version of the paper. The omitted proofs are contained in [10].

1.1 Preliminaries and Notation

An *instance* σ of the on-line load balancing problem with processors P and task types $\mathcal{T}$ is defined as a sequence of $\texttt{new}(\cdot,\cdot)$ and $\texttt{del}(\cdot)$ commands. In particular: (i) $\texttt{new}(x,w)$ means that a new task of weight w and type $x \in \mathcal{T}$ is created; (ii) $\texttt{del}(i)$ means that the task created by the i-th $\texttt{new}(\cdot,\cdot)$ command of the instance is deleted.

As already mentioned, we model the problem by means of a bipartite graph $G_{P,\mathcal{T}}(X_{\mathcal{T}} \cup P, E_{\mathcal{T}})$, where $\mathcal{T}$ depends on the problem version we are considering. For the sake of brevity, in the following we will always omit the subscripts '$P,\mathcal{T}$' and '$\mathcal{T}$' since the set of processors and the set of task types will be clear from the context. Given a graph $G(V,E)$, $\Gamma_G(v)$ denotes the *open* neighborhood of the node $v \in V$. So, a task of type x can be assigned to $\Gamma_G(x)$.

We will distinguish between the *unweighted* case, in which all tasks have weight 1, and the *weighted* case, in which the weights may vary from task to task. We also refer to (un-)weighted tasks to denote these variants.

Given an instance σ, a *configuration* is an assignment of the tasks of σ to the processors in P, such that each task t is assigned to a processor in P_t. Given a configuration C, we denote with $l_C(i)$ the *load* of processor p_i, that is, the sum of the weights of all tasks assigned to it. In the sequel, we will usually omit the configuration when it will be clear from the context. The load of C is defined as the maximum of all the processor loads and is denoted with $l(C)$. Given an instance $\sigma = \sigma_1 \cdots \sigma_n$ and given an on-line algorithm $\mathcal{A}$, let $C_h^{\mathcal{A}}$ be the configuration reached by $\mathcal{A}$ after processing the first h commands. Moreover, let C_h^{off} be the configuration reached by the optimal off-line algorithm after processing the first h commands. Let also $\texttt{opt}(\sigma) = \max_{1 \le h \le n} l(C_h^{\text{off}})$ and $l_{\mathcal{A}}(\sigma) = \max_{1 \le h \le n} l(C_h^{\mathcal{A}})$.

An on-line algorithm $\mathcal{A}$ is said to be *r-competitive* if there exists a constant b such that, for any instance σ, it holds that $l_{\mathcal{A}}(\sigma) \le r \cdot \texttt{opt}(\sigma) + b$. An on-line algorithm $\mathcal{A}$ is said to be *strictly r-competitive* if, for any instance σ, it holds that $l_{\mathcal{A}}(\sigma) \le r \cdot \texttt{opt}(\sigma)$.

A simple on-line algorithm for the load-balancing problem described above is the *greedy algorithm* that assigns a new task to the least loaded processor among those processors that can serve the task. That is, whenever a $\texttt{new}(x,w)$ command is encountered and the current configuration is C, the greedy algorithm looks for the processor p_i in $P_{t=(x,w)}$ such that $l_C(i)$ is minimal and assigns the new task $t = (x,w)$ to p_i. (Ties are broken arbitrarily.)

2 (Sub-)graphs and (Sub-)greedy Algorithms

In the sequel we will describe an on-line load balancing algorithm whose competitive ratio depends on some combinatorial properties of $G(X \cup P, E)$. The two main ideas used in our approach are the following:

1. We remove some edges on G and then we apply the greedy algorithm *to the resulting bipartite graph*;
2. While removing edges we try to balance the number of processors used by tasks of type $x \in X$ and the number of processors that the adversary can use to assign the same set of tasks in the original graph G.

First of all, let us observe that our method aims to obtain a good competitive ratio by *adding further constraints* to the original problem: this indeed corresponds to remove a suitable set of edges from G. Choosing which edges to remove depends on some combinatorial properties we want the resulting bipartite graph to satisfy. Before giving a formal description of such properties, we will describe the basic idea behind our approach.

The main idea. Let us consider a generic iteration of an algorithm that has to assign a task of type $x \in X$. Assume that our algorithm takes into account a set $U(x)$ of processors and assigns the task to the least loaded one. In order to evaluate the competitive ratio of this approach we need to know which set of processors $A(x)$ an adversary can use to assign the overall load currently in $U(x)$ (as we will see in the sequel the competitive ratio of our algorithm is roughly $|A(x)|/|U(x)|$). In the following, we will show how the set $A(x)$ is determined by the choices of our algorithm in the previous steps (see Fig. 2).

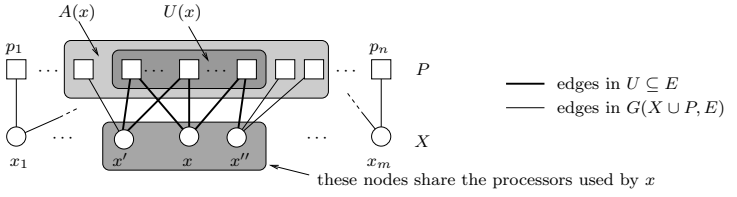

Fig. 2. The main idea of the sub-greedy algorithm is to balance the ratio between the number of used processors $|U(x)|$ and the number of processors available $|A(x)|$ to the adversary.

2.1 Analysis

In this section we formalize the idea above and we provide the performance analysis of the resulting algorithm.

Definition 1 (Used Processors). *For any $x \in X$, we define a non-empty set $U(x) \subseteq \Gamma_G(x)$ of* used processors*. Moreover, given a processor $p \in P$, we denote by $U^{-1}(p)$ those vertices in X that have p as an available processor, i.e. $U^{-1}(p) = \{x \mid p \in U(x)\}$.*

Definition 2 (Adversary Processors). *For any $x \in X$ we denote by $A(x)$ those processors that an off-line adversary can use to balance the load assigned to $U(x)$. In particular, $A(x) = \bigcup_{p \in U(x)} \bigcup_{x' \in U^{-1}(p)} \Gamma_G(x')$.*

Notice that the set $U(x)$ specifies a subset of the edges in $G(X \cup P, E)$ incident to x. By considering the union over all $x \in X$ of these edges and the resulting bipartite subgraph, we have the following:

Definition 3 (Sub-Greedy Algorithm). *For any bipartite graph $G(X \cup P, E)$ and for any subset of edges $U \subseteq E$, the* sub-greedy algorithm *is defined as the greedy on-line algorithm applied to $G_U = G(X \cup P, U)$.*

Remark 1. It is easy to see that the sub-greedy algorithm is a special case of the cluster algorithm in [9], since the latter imposes each connected component of G_U to be a complete bipartite graph [9, Definition of Cluster].

It is clear that the performance of the sub-greedy algorithm will depend on the choice of the set $U \subseteq E$. In particular, we can characterize its competitive ratio in terms of the ratio between the set of adversary processors $A(x)$ and the set of used processors $U(x)$. Let us consider the following quantities:

$$\rho_{\mathsf{w}}(U) = \max_{x \in X} \left\{ \frac{|A(x)| - 1}{|U(x)|} \right\}, \qquad \rho_{\mathsf{u}}(U) = \max_{x \in X} \left\{ \frac{|A(x)|}{|U(x)|} \right\}.$$

Then, the following two results hold.

Theorem 1. *The sub-greedy algorithm is strictly $(1 + \rho_{\mathsf{w}}(U))$-competitive in the case of weighted tasks.*

Proof. Let p_i be the processor with the highest load and let $t = (w, x)$ be the last task assigned to p_i by the sub-greedy algorithm. Since t has been assigned to p_i whose load, before the arrival of t, was $l(i) - w$, we have that each processor in $U(x)$ had load at least $l(i) - w$. So, the overall load of $U(x)$ is at least $|U(x)|(l(i) - w) + w$. We now consider the number of processors that any off-line strategy can use to spread such load. Such a number is equal to $|A(x)|$, which implies that the optimal off-line solution has measure at least

$$l^* \geq \max \left\{ \frac{|U(x)|l(i) - w(|U(x)| - 1)}{|A(x)|}, w \right\}.$$

The worst case is when we equate the two quantities, that is $l^* \geq w = \frac{|U(x)|l(i)}{|A(x)| + |U(x)| - 1}$, which implies the following bound on the competitive ratio

$$\frac{l(i)}{l^*} \leq \frac{|A(x)| + |U(x)| - 1}{|U(x)|} \leq 1 + \rho_{\mathsf{w}}(U).$$

Hence, the theorem follows.

Theorem 2. *The sub-greedy algorithm is $\rho_{\mathsf{u}}(U)$-competitive (resp., strictly $\lceil \rho_{\mathsf{u}}(U) \rceil$-competitive) in the case of unweighted tasks.*

Proof. Let us consider a generic iteration of the sub-greedy algorithm in which a task t arising in $x \in X$ has been assigned to $p_i \in U(x)$. Since t has been assigned to p_i whose load, before the arrival of t, was $l(i)-1$, we have that each processor in $U(x)$ had load at least $l(i)-1$. This implies that the overall number of tasks in $U(x)$, after the arrival of t, was at least $|U(x)|(l(i)-1)+1$. Let us also observe that the number of processors to which the off-line optimal solution can assign these tasks is at most $|A(x)|$. Thus, the optimal off-line solution has measure at least

$$l^* \geq \frac{|U(x)|(l(i)-1)+1}{|A(x)|} \geq \frac{l(i)-1}{\rho_{\mathsf{u}}(U)} + \frac{1}{|A(x)|}. \tag{1}$$

By contradiction, let us suppose that $l(i) > l^* \lceil \rho_{\mathsf{u}}(U) \rceil$. Then, since both $l(i)$ and l^* have integer values, we have that $l(i) - 1 \geq l^* \lceil \rho_{\mathsf{u}}(U) \rceil \geq l^* \rho_{\mathsf{u}}(U)$. This leads to the following contradiction:

$$l^* \geq \frac{l(i)-1}{\rho_{\mathsf{u}}(U)} + \frac{1}{|A(x)|} \geq \frac{l^* \rho_{\mathsf{u}}(U)}{\rho_{\mathsf{u}}(U)} + \frac{1}{|A(x)|} > l^*.$$

We have thus proved that the sub-greedy algorithm is strictly $\lceil \rho_{\mathsf{u}}(U) \rceil$-competitive. Finally, Eq. 1 implies

$$l(i) < l^* \rho_{\mathsf{u}}(U) + \frac{1}{\rho_{\mathsf{u}}(U)} \leq l^* \rho_{\mathsf{u}}(U) + 1,$$

where the last inequality follows from the fact that $\rho_{\mathsf{u}}(U) \geq 1$. So, the sub-greedy algorithm is also $\rho_{\mathsf{u}}(U)$-competitive. Hence, the theorem follows.

We next show the limits of our approach as it is and we generalize it in order to handle more cases. First, consider the bipartite graph $G(X \cup P, E)$ with $X = \{x_1, x_2, \ldots, x_n\} \cup \{x_0\}$ and $E = \{(x_i, p_i) |\ 1 \leq i \leq n\} \cup \{(x_0, p_i) |\ 1 \leq i \leq n\}$. It is easy to see that any subset of edges U yields $\rho_{\mathsf{w}}(U) = n - 1$. However, a rather simple idea might be to separate the high-degree vertex x_0 from the low-degree vertices $x_1, x_2, \ldots, x_n$. So, tasks of type x_0 are processed *independently* from tasks of type $x_1, x_2, \ldots, x_n$. It is possible to prove that this algorithm has a constant competitive ratio. This idea leads to the following:

Definition 4 (sub-greedy*). *Let $X_1, X_2, \ldots, X_k$ be any partition of the set X of the task types vertices of $G(X \cup P, E)$. Also let $G_i = G(X_i \cup P, E_i)$ be the corresponding induced subgraph, and let $U_i \subseteq E_i$ for $1 \leq i \leq n$. We denote by* sub-greedy* *the algorithm assigning tasks of type in X_i as the sub-greedy algorithm on the subgraph of G_i corresponding to U_i, and with only these tasks as input (i.e., independently of tasks of other types).*

In the sequel we denote by $\rho_{\mathsf{w}}(U_i, G_i)$ the quantity $\rho_{\mathsf{w}}(U)$ computed w.r.t. the graph G_i and subset of edges $U_i \subseteq E_i$.

Theorem 3. *The sub-greedy* algorithm is strictly* $(k + \rho^*_{\mathsf{w}}(U))$*-competitive in the case of weighted tasks, where* k *is the number of subgraphs and* $\rho^*_{\mathsf{w}}(U) = \sum_{i=1}^{k} \rho_{\mathsf{w}}(U_i, G_i)$.

Proof. Given a sequence of tasks σ, let $\sigma(i)$ denote the subsequence containing tasks whose type is in X_i. Also let $l(j)$ denote the load of processor p_j at some time step and $l^i(j)$ the load at the same time step w.r.t. tasks corresponding to X_i only. Then, the definition of sub-greedy* and Theorem 1 imply $\max_{1\le j\le n} l^i(j) \le \mathsf{opt}(\sigma(i))(1 + \rho_{\mathsf{w}}(U_i, G_i))$, for $1 \le i \le k$. It then holds that

$$\max_{1\le j\le n} l(j) \le \sum_{i=1}^{k} \max_{1\le j\le n} l^i(j) \le \sum_{i=1}^{k} \mathsf{opt}(\sigma(i))(1 + \rho_{\mathsf{w}}(U_i, G_i)) \tag{2}$$

$$\le k \cdot \mathsf{opt}(\sigma) + \mathsf{opt}(\sigma) \sum_{i=1}^{k} \rho_{\mathsf{w}}(U_i, G_i), \tag{3}$$

where the last inequality follows from the fact that $\mathsf{opt}(\sigma(i)) \le \mathsf{opt}(\sigma)$.

The above theorem will be a key-ingredient in deriving algorithms for the general case (see Sect. 4).

2.2 Computing Good Subgraphs

From Theorems 1-2 it is clear that, in order to attain a good competitive ratio, it is necessary to select a subset of edges $U \subseteq E$ such that $\rho(U)$ is as small as possible. Similarly, Theorem 3 implies that U should minimize $k + \rho^*_{\mathsf{w}}(U)$, when considering the sub-greedy* algorithm.

We now rewrite the sets $A(x)$ and $U(x)$ by looking at the *open* neighborhood operator $\Gamma_G(\cdot)$. In particular, we have that $U(x) = \Gamma_{G_U}(x)$ and $A(x) = \Gamma_G(\Gamma_{G_U}(\Gamma_{G_U}(x)))$. When considering the weighted case, this leads to the following optimization problem:

Problem 5 *Min Weighted Adversary Subgraph (MWAS).*
Instance: *A bipartite graph* $G(X \cup P, E)$.
Solution: *A subgraph* $G_U = G(X \cup P, U)$*, such that* $U \subseteq E$ *and, for every* $x \in X$, $|\Gamma_{G_U}(x)| \ge 1$.
Measure: $\rho_{\mathsf{w}}(U, G) = \max_{x\in X} \frac{|\Gamma_G(\Gamma_{G_U}(\Gamma_{G_U}(x)))|-1}{|\Gamma_{G_U}(x)|}$.

Problem 6 *Min Weighted Adversary Multi-Subgraph (MWAMS).*
Instance: *A bipartite graph* $G(X \cup P, E)$.
Solution: *A partition* $X_1, X_2, \ldots, X_k$ *of* X *and a collection* $U = \{U_1, \ldots, U_k\}$ *of subsets of edges* $U_i \subseteq E_i$*, where* E_i *denotes the set of edges of the subgraph* $G_i = G(X_i \cup P, E_i)$ *induced by* X_i*, such that, for every* $1 \le i \le k$ *and* $x \in X_i$, $|\Gamma_{G_{U_i}}(x)| \ge 1$.
Measure: $k + \rho^*_{\mathsf{w}}(U, G) = k + \sum_{i=1}^{k} \rho_{\mathsf{w}}(U_i, G_i)$.

Similarly, the Min Unweighted Adversary Subgraph (MUAS) problem and the Min Unweighted Adversary Multi-Subgraph (MUAMS) problem are defined by replacing ρ_{w} with ρ_{u} in the two definitions above, respectively.

It is possible to construct a reduction showing the NP-hardness of all these problems (see [10]). Also, the same reduction is a gap-creating reduction, thus implying the non existence of a PTAS for any such problem. In particular, we obtain the following result:

Theorem 4. *The MUAS and MWAS problems cannot be approximated within a factor smaller than* $7/6$ *and* $3/2$*, respectively, unless* $\mathsf{P} = \mathsf{NP}$*. Moreover MUAMS and MWAMS cannot be approximated within a factor smaller than* $11/10$ *and* $5/4$*, respectively, unless* $\mathsf{P} = \mathsf{NP}$*.*

3 Application to Hierarchical Server Topologies

In this section we apply our method to the hierarchical server topologies introduced in [7]. In particular, we consider the *linear* hierarchical topology: Processors are ordered from p_1 (the most capable processor) to p_n (the least capable processor) in decreasing order with respect to their capabilities. So, if a task can be assigned to processor p_i, for some i, then it can also be assigned to any p_j with $1 \leq j < i$. We can therefore consider task types corresponding to the intervals $\{p_1, p_2, \ldots, p_i\}$, for each $1 \leq i \leq n$.

The resulting bipartite graph $G(X \cup P, E)$ is given by $X = \{x_1, \ldots, x_n\}$, $P = \{p_1, \ldots, p_n\}$ and $E = \{(x_i, p_j) \mid x_i \in X, p_j \in P, j \leq i\}$. We denote this graph as K_n^{hst}.

We next provide an efficient construction of subgraphs of K_n^{hst}.

Lemma 1. *For any positive integer* n*, there exists a* U *such that* $\rho_{\mathsf{w}}(U, K_n^{hst}) \leq (4n-2)/(n+2)$ *and* $\rho_{\mathsf{u}}(U, K_n^{hst}) \leq (4n)/(n+2)$*. Moreover, the set* U *can be computed in linear time.*

Proof. For each $1 \leq i \leq n$, we define the set $U(x_i)$ as

$$U(x_i) = \{p_{\lceil i/2 \rceil}, p_{\lceil i/2 \rceil + 1}, \ldots, p_i\} = \begin{cases} \{p_{i/2}, p_{i/2+1}, \ldots, p_i\} & \text{if } i \text{ is even,} \\ \{p_{(i+1)/2}, p_{(i+1)/2+1}, \ldots, p_i\} & \text{otherwise.} \end{cases}$$

Clearly, $|U(x_i)| = i/2 + 1$ if i is even, and $|U(x_i)| = (i+1)/2$ otherwise. Moreover, $|A(x_i)| = \max_{i \leq j \leq n}\{j \mid U(x_i) \cap U(x_j) \neq \emptyset\}$. It is easy to see that $|A(x_i)| \leq 2i$, thus implying

$$\rho_{\mathsf{w}}(U) = \max_{1 \leq i \leq n} \frac{|A(x_i)| - 1}{|U(x_i)|} \leq \max_{1 \leq i \leq n} \frac{4i-2}{i+1} \leq \frac{4n-2}{n+1}.$$

A better bound can be obtained by distinguishing two cases: 1) $i \leq n/2$ and 2) $i \geq n/2 + 1$. In case 1) we apply the bound above, while for 2) we simply use $|A(i)| \leq n$; in both cases we obtain $\rho_{\mathsf{w}}(U) \leq (4n-2)/(n+2)$.

With a similar proof we can show that the same construction yields $\rho_{\mathsf{u}}(U) \leq (4n)/(n+2)$.

An immediate application of Lemma 1 combined with Theorems 1-2 is the following result:

Theorem 5. *For linear hierarchy topologies the sub-greedy algorithm is strictly* $(5n)/(n+2)$*-competitive in the case of weighted tasks, and* $(4n)/(n+2)$*-competitive and strictly* 4*-competitive for unweighted tasks.*

Notice that our approach improves over the 5-competitive (respectively, 4-competitive) algorithm for weighted (respectively, unweighted) tasks given in [7].

Remark 2. Observe that, if we impose the subgraph to be a set of complete bipartite graphs as in [9], then K_n^{hst} does not admit a construction yielding $O(1)$-competitive algorithms. So, for these topologies, the sub-greedy algorithm constitutes a significant improvement w.r.t. the result in [9].

4 The General Case

In this section we provide a sufficient condition for obtaining $O(\sqrt{n})$-competitive algorithms. This applies to the hierarchical server model when the order of the servers is a tree. Thus, in this case our result is optimal because of the $\Omega(\sqrt{n})$ lower bound [7].

We first define an overall strategy to select the set $U(x)$ depending on the degree $\delta(x)$ of x:

High degree (easy case): $\delta(x) \geq \sqrt{n}$**.** In this case we use *all of its adjacent vertices in* P. Since $|U(x)| = \delta(x) \geq \sqrt{n}$, we have $|A(x)|/|U(x)| \leq \sqrt{n}$.

Low degree (hard case): $\delta(x) < \sqrt{n}$**.** For low degree vertices our strategy will be to choose a *single* processor p_x^* in $\Gamma_G(x) \subseteq P$. The choice of this element must be carried out carefully so to guarantee $|A(x)| \leq \sqrt{n}$. For instance, it would suffice that p_x^* does not appear in any other set $U(x')$.

Then, our next idea will be to partition the graph $G(X \cup P, E)$ into two subgraphs $G_l(X_l \cup P, E_l)$ and $G_h(X_h \cup P, E_h)$ containing low and high degree vertices, respectively. Notice that, if we are able to have a $f(n)$-competitive algorithm for the low degree graph, then we have a $O(\sqrt{n} + f(n))$-competitive algorithm for our problem (see Theorem 3).

We next focus on low degree graphs and we provide sufficient conditions for $O(\sqrt{n})$-competitive algorithms.

Theorem 6. *If* $G_l(X_l \cup P, E_l)$ *admits a* b*-matching, then the sub-greedy* algorithm is at most* $((b+1)\sqrt{n}+2)$*-competitive.*

Proof. Let U be a b-matching for G_l. It is easy to see that in G_l $|A(x)| \leq b\sqrt{n}$, for all $x \in X_l$. Thus, $\rho_{\mathsf{w}}(U, G_l) \leq b\sqrt{n}$. By definition of G_h, $\rho_{\mathsf{w}}(E_h, G_h) \leq \sqrt{n}$. We can thus apply Theorem 3 with $k = 2$, $U_1 = U$ and $U_2 = E_h$. Hence the theorem follows.

Theorem 7. *If $G(X \cup P, E)$ admits a b-matching, then the sub-greedy* algorithm is at most $(2\sqrt{bn}+2)$-competitive.*

Proof Sketch. Define low-degree vertices as those $x \in X$ such that $\Gamma_G(x) \leq \sqrt{n/b}$. Subgraphs G_l and G_h are defined accordingly. The existence of a b-matching U yields $\rho_{\mathsf{w}}(U, G_l) \leq b\sqrt{n/b}$. By definition of G_h, $\rho_{\mathsf{w}}(E_h, G_h) \leq \sqrt{bn}$. We can thus apply Theorem 3 with $k = 2$, $U_1 = U$ and $U_2 = E_h$. Hence the theorem follows.

Theorem 8. *If $G(X \cup P, E)$ admits a matching, then the sub-greedy algorithm is at most $\delta_{\max}$-competitive, where $\delta_{\max} = \max_{x \in X} |\Gamma_G(x)|$.*

Proof. Let U be a matching for G. Then, $|A(x)| \leq |\Gamma_G(x)|$ and $|U(x)| = 1$, for any $x \in X$.

4.1 Generalized Hierarchical Server Topologies

We now apply these results to the hierarchical model in the case in which the ordering of the servers forms a tree. Figure 1 shows an example of this problem version: processors are arranged on a rooted tree and there is a task type x_i for each node p_i of the tree; a task of type x_i can be assigned to processor p_i or to any of its ancestors.

We first generalize this problem version to a more general setting:

Definition 7. *Let $H(P, F)$ be a directed graph. The associated bipartite graph $G_H(X \cup P, E)$ is defined as $X = \{x_1, x_2, \ldots, x_n\}$, and $(x_i, p_j) \in E$ if and only if $i = j$ or there exists a directed path in H from p_i to p_j.*

We can model a tree hierarchy by considering a rooted tree T whose edges are directed *upward*. We then obtain the following:

Theorem 9. *For any rooted tree $T(P, E)$ the corresponding graph $G_T(X \cup P, E)$ admits a matching. In this case, the sub-greedy* algorithm is always at most $(2\sqrt{n}+2)$-competitive. Moreover, the sub-greedy algorithm is at most h-competitive, where h is the height of T.*

Proof. It is easy to see that $M = \{(x_i, p_i) | \ \ 1 \leq i \leq n\}$ is a matching for $G_T(X \cup P, E)$. We can thus apply Theorem 6 with $b = 1$.

Theorem 10. *Let $H(P, F)$ be any directed graph representing an ordering among processors, and let $G_H(X \cup P, E)$ be the corresponding bipartite subgraph. Then, the sub-greedy* algorithm is at most $(2\sqrt{n}+2)$-competitive.*

Proof Sketch. We first reduce every strongly connected component of H to a single vertex, since processors of this component are equally powerful: if a task can be assigned to a processor of this component, then it can also be assigned to any other processor of the same component. (Equivalently, this transformation

does not affect G_H.) So, we can assume H being not acyclic. We then greedily construct a matching U by repeating the following three steps: 1) for a p_i with no outgoing edges, include the edge (x_i, p_i) in U; 2) remove p_i and x_i from both H and G_H.

Since H is acyclic, such a vertex p_i must exist. Moreover, in G_H, p_i is adjacent to x_i only (otherwise, p_i must have one out-going edge in H). Removing p_i and x_i from G_H yields the graph corresponding to $H \setminus \{p_i\}$. After step 3), we are left with a new H' and $G_{H'}$ which enjoys the same property as H. So, we can iterate this procedure until all vertices in H are removed. Since the number of task types equals the number of vertices of H, this method yields a matching for G_H.

5 Conclusions and Open Problems

We have presented a novel technique which allows to derive on-line algorithms with a simple modification of the greedy one. This modification preserves the good feature of deciding where to assign a task solely based on the current load of processors to which that task can be potentially assigned to. Indeed, the pre-computation of the subgraph required by our approach is performed off-line given the graph representing the problem constraints. Additionally, for several cases we have considered here, this subgraph is only used in the analysis, while the resulting algorithms are simple modifications of the greedy *implementing* the subgraph: the construction of Lemma 1 yields an algorithm performing a greedy choice on the rightmost half of the available processors $\Gamma_G(x_i) = \{p_1, \ldots, p_i\}$. So, this algorithm can be implemented even without knowing n. A similar argument applies to the sub-greedy* algorithm with the subgraph of Theorem 9: in this case knowing n is enough to decide whether a vertex as "low-degree" or not; in the latter case the matching (x_i, p_i) yields a fixed assignment for tasks corresponding to type x_i.

In general, the adopted strategy of the sub-greedy* algorithm depends on the type of the task and on the current load of the adjacent processors in the appropriate subgraph. Since the algorithm assigns tasks corresponding to different subgraphs *independently*, it must be able to compute the load of a processor w.r.t. a subset X_i; this can be easily done whenever tasks are specified as pairs (x, w).

So, our algorithms are distributed and, for the generalized hierarchical topologies, their competitive ratio is only slightly worse than the $2\sqrt{n}+1$ upper bound provided by the Robin-Hood algorithm [5]. Also, for tree hierarchical topologies, our analysis yields a much better ratio whenever the height h of the tree is $o(\sqrt{n})$ (e.g., for balanced trees).

An interesting direction for future research might be that of characterizing the competitive ratio of distributed algorithms under several assumptions on the graph $G(X \cup P, E)$: 1) G is *unknown*, 2) G is uniquely determined by n, but n is unknown, 3) G is known.

A related question is: under which hypothesis does our technique yield optimal competitive ratios?

Acknowledgements. The fourth author wishes to thank Amotz Bar-Noy for a useful discussion and for bringing the work [7] to his attention.

References

1. S. Albers. Better bounds for on-line scheduling. *Proc. of the 29th ACM Symp. on Theory of Computing (STOC)*, pages 130–139, 1997.
2. Y. Azar. *On-line load balancing*, chapter in "On-line Algorithms - The state of the Art", A. Fiat and G. Woeginger (eds.). Springer Verlag, 1998.
3. Y. Azar, A. Broder, and A. Karlin. Online load balancing. *Theoretical Computer Science*, 130:73–84, 1994.
4. Y. Azar and L. Epstein. On-line load balancing of temporary tasks on identical machines. *Proc. of the 5th Israeli Symposium on Theory of Computing and Systems (ISTCS)*, pages 119–125, 1997.
5. Y. Azar, B. Kalyanasundaram, S. Plotkin, K. Pruhs, and O. Waarts. Online load balancing of temporary tasks. *Journal of Algorithms*, 22:93–110, 1997.
6. Y. Azar, J. Naor, and R. Rom. The competitiveness of online assignments. *Journal of Algorithms*, 18:221–237, 1995.
7. A. Bar-Noy, A. Freund, and J. Naor. On-line load balancing in a hierarchical server topology. *SIAM Journal on Computing*, 31(2):527–549, 2001. Preliminary version in Proc. of the 7th Annual European Symposium on Algorithms, ESA'99.
8. D. Braess. Ueber ein Paradoxon aus der Verkehrsplanung. *Unternehmensforschung*, 12:258–268, 1968.
9. P. Crescenzi, G. Gambosi, and P. Penna. On-line algorithms for the channel assignment problem in cellular networks. In *Proc. of the 4th ACM International Workshop on Discrete Algorithms and Methods for Mobile Computing (DIALM)*, pages 1–7, 2000. Full version to appear in Discrete Applied Mathematics.
10. P. Crescenzi, G. Gambosi, and P. Penna. On-line load balancing made simple: Greedy strikes back. Technical report, Università di Salerno, 2003. Electronic version available at `http://www.dia.unisa.it/~penna`.
11. R. Graham. Bounds for certain multiprocessor anomalies. *Bell System Technical Journal*, 45:1563–1581, 1966.
12. R. Graham. Bounds on multiprocessor timing anomalies. *SIAM J. Appl. Math.*, 17:263–269, 1969.
13. S. Guha and S. Khuller. Greedy strikes back: Improved facility location algorithms. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 1998.
14. E. Tardòs J.K. Lenstra, D.B. Shmoys. Approximation algorithms for scheduling unrelated parallel machines. *Math. Programming*, 46:259–271, 1990.
15. J. D. Murchland. Braess's paradox of traffic flow. *Transportation Research*, 4:391–394, 1070.
16. S. Phillips and J. Westbrook. Online load balancing and network flow. *Algorithmica*, 21(3):245–261, 1998.
17. Serge Y. Ma and A. Plotkin. An improved lower bound for load balancing of tasks with unknown duration. *Information Processing Letters*, 62(6):301–303, 1997.