ELSEVIER

Available at
www.ElsevierMathematics.com
POWERED BY SCIENCE DIRECT®

Discrete Applied Mathematics 137 (2004) 237–266

DISCRETE
APPLIED
MATHEMATICS

www.elsevier.com/locate/dam

On-line algorithms for the channel assignment problem in cellular networks

Pilu Crescenzi[a,1], Giorgio Gambosi[b], Paolo Penna[b]

[a] *Dipartimento di Sistemi e Informatica, Università di Firenze, via C. Lombroso 6/17, I-50134 Firenze, Italy*

[b] *Dipartimento di Matematica, Università di Roma "Tor Vergata", via della Ricerca Scientifica, I-00133 Roma, Italy*

Received 21 April 2001; received in revised form 18 July 2002; accepted 24 February 2003

Abstract

We consider the on-line channel assignment problem in the case of cellular networks and we formalize this problem as an on-line load balancing problem for temporary tasks with restricted assignment. For the latter problem, we provide a general solution (denoted as the *cluster algorithm*) and we characterize its competitive ratio in terms of the combinatorial properties of the graph representing the network. We then compare the cluster algorithm with the greedy one when applied to the channel assignment problem: it turns out that the competitive ratio of the cluster algorithm is strictly better than the competitive ratio of the greedy algorithm. The cluster method is general enough to be applied to other on-line load balancing problems and, for some topologies, it can be proved to be optimal.

© 2003 Elsevier B.V. All rights reserved.

1. Introduction

In this paper, we consider the on-line channel assignment problem in the case of cellular systems, which is defined as follows. A given set of mobile users has to be assigned to a given set of available cells: The assignment of each user depends on the topology of the network and on the position of the user. Each user can move itself from

E-mail addresses: piluc@dsi.unifi.it (P. Crescenzi), gambosi@mat.uniroma2.it (G. Gambosi), penna@mat.uniroma2.it (P. Penna).

[1] Research partially supported by Italian MURST project "Algoritmi per Grandi Insiemi di Dati: Scienza ed Ingegneria".

0166-218X/$ - see front matter © 2003 Elsevier B.V. All rights reserved.
doi:10.1016/S0166-218X(03)00341-X

one position to another or it can "terminate" its request (thus disappearing from the set of users). Additionally, users may require, for their requests, different bandwidths corresponding to different types of services (such as video and/or audio applications or file transfers). The number of users assigned to the same cell clearly affects the number of frequencies used by that cell in order to satisfy all the requests, since the frequencies used by the cell must be at a minimum separation distance, usually greater than two (see, for instance, [23,24]). For this reason, it is important to minimize the maximum cell load (i.e., the maximum number of users assigned to the same cell) among all the cells.

The off-line version of this problem has been already considered in [19]: in that paper, the topology of the cellular system has been exploited in order to provide a characterization of the instances (i.e., set of users along with their positions) that admit a solution, that is, an assignment to the cells that does not exceed the network capacity. However, the proposed solution requires that several (potentially, *all*) users have to be reassigned whenever a new one arrives. This is clearly infeasible from a practical point of view. So, up to our knowledge, no efficient on-line strategy to assign users to the cells has been presented before. Moreover, only the "unweighted" restriction of the problem is considered, i.e., the case in which each user can request only a globally fixed resource (bandwidth or frequency). The on-line frequency assignment problem [2] has been the subject of several works (see for instance [9,12,16,20]). This problem consists of assigning frequencies to users so that interference constraints are satisfied (e.g., two users within adjacent cells cannot use the same frequency). The goal is to minimize the span, that is, the difference between the largest and the smallest frequencies used. It is worth observing that the frequency assignment problem is only related in as far as it is the problem to be solved after users have been assigned to cells.

In this paper, we first observe (see Section 2) that the channel assignment problem has a very natural formulation as an 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.

The problem asks to find an assignment of the tasks to the processors which minimizes the maximum processor load (that is, the maximum sum of the costs of all tasks assigned to the same processor) among all processors. Observe that coping with mobile "unpredictable" users is, indeed, one of the major motivations for studying on-line load balancing. Moreover, the idea of balancing the load within a cellular network in order to optimize the use of the available frequencies already appeared in [10]. Several variants of the above described on-line load balancing problem have already

[2] This problem is sometimes referred to as the channel assignment problem in the literature. We instead use the term "frequency assignment" to distinguish it from our problem.

Table 1
Competitiveness results for cellular networks

Greedy	Our algorithm	Lower bound
At least 5 (Theorem 4.3)	4 (Theorem 4.2)	3 (Theorem 4.4)

been studied in the literature [1,3–6,22,25] (see also [2] for a survey). For example, an optimal algorithm for the more general case, in which the subset of processors a task may be assigned to is the entire set of processors, has been proposed in [5]. This algorithm is $(2\sqrt{n}+1)$-competitive, where n is the number of processors, and it has been proved that this performance is optimal up to a constant factor [3]: intuitively, 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. Clearly, this algorithm can be applied to our problem but we cannot hope to attain a better competitive ratio if we do not exploit some information on the specific problem we are considering. After all, a mobile user cannot be potentially assigned to any base station on the earth surface!

The main contribution of this paper (see Section 3) is to provide a general solution to the on-line load-balancing problem which takes into account and exploits certain properties of the constraints of the problem. In particular, we introduce a graph-based model to formalize the problem and we describe an algorithm (denoted as the *cluster algorithm*) whose competitiveness is characterized by some combinatorial properties of the input graph. The main idea of our approach is to *add further constraints to the original problem* and then apply a simple greedy technique to the resulting new problem. A first obvious advantage of our approach is that it maintains the simplicity of the greedy algorithm. Moreover, the method results in a significant reduction on the communication among nodes of the network thus making the approach particularly suitable in a distributed setting (such as the mobile one).

We apply the cluster method to the specific case of interest in the mobile context, that is, the channel assignment problem in the case of cellular networks (see Section 4 and Table 1). In particular, we prove that our algorithm is 4-competitive both in the case of arbitrary weighted requests and in the case of unitary weight requests, i.e., in the case in which all the requests have bandwidth equal to 1.[3] We also show that the simple greedy approach is at least 5-competitive (a performance strictly worse than the cluster algorithm) and that no algorithm can be less than 3-competitive. Hence, the cluster algorithm is not so far from being optimal. Both these latter results hold also in the unitary weight case. The additional advantage of our algorithm, when applied to cellular topologies, is that it reduces the communication among cells: indeed, we will see that, once a user request arrives, it can be assigned to the "right" cell *without querying any cell* about its current load.

[3] The restriction to unitary weights is clearly representative of all the cases in which the weights have the same value and this value is a constant *independent* of the instance.

We also apply the cluster algorithm to a simple one-dimensional topology and, once again, we compare our approach with the greedy technique (see Section 5). This topology is certainly a simplification of what may happen in the reality (even though it has been already considered in [18] for broadcasting problems): however, the problem is still non-trivial. Moreover, it constitutes another example for which it can be proved that the simple greedy algorithm is not optimal and that the cluster algorithm performs better than the greedy one.

Finally, we mention that our approach may have several applications in satellite cellular systems, where channel capacity among satellites becomes a crucial aspect, since it takes into account the problem of handover due to satellites' movement in low Earth orbit constellations.

2. From channel assignment to load balancing

In the cellular network channel assignment problem, we are given a set of two-dimensional circular *cells* (also called *base stations*) which overlap as shown in Fig. 1(a); actually, the simpler graphical representation shown in Fig. 1(b) will be used throughout the paper. Observe that any point of the two-dimensional space belongs to at most three cells: in the simplified representation, a point p belongs to three cells if it coincides with their common vertex, p belongs to two cells if it lies on their common edge, and p belongs to one cell if it lies in its interior. Observe also that this setting is a special case of the one in which the overlap between cells can be arbitrary (see e.g. [19]). This special case has been studied in the case of frequency assignment problems (see e.g. [9,12,16,20,23,24]). A *communication request* r is specified by the point p_r of the two-dimensional space where the request arises and the bandwidth b_r required by the request. Whenever a communication request r arises, it must be served by one of the base stations that include p_r. Clearly, if p_r belongs to one cell only, then there is no choice. But if p_r belongs to the intersection of two or three cells, then r can be served by any of the intersecting cells. Communication requests can *move* themselves from one point to another: we simulate this phenomenon by assuming that, whenever a request crosses the border of a cell, then it "dies" and a new request arises in the new point with the same bandwidth of the original one. The *channel assignment problem* then consists of serving all the arising requests while minimizing, at any instant, the maximum cell load (i.e., the maximum sum of the bandwidths of all the active requests served by the same cell) among all cells.

If we view each cell as a processor and each communication request as a task, the channel assignment problem can be formulated as an on-line load-balancing problem in the case of temporary tasks with restricted assignment and no preemption. Let us first define such a problem and subsequently show how the channel assignment problem can be formulated in its terms.

2.1. The on-line load-balancing problem

Let $P = \{p_1, \ldots, p_n\}$ be a set of processors and let $\mathscr{T} \subseteq 2^P$ be a set of *task types*. We represent the set of task types by means of an *associated bipartite graph*

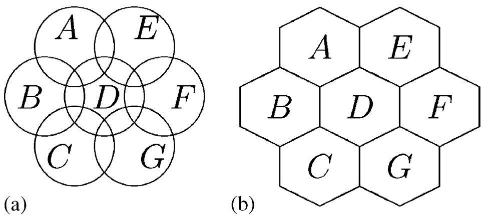


Fig. 1. A set of base stations and its graphical representation.

$G_{P,\mathscr{T}}(X_{\mathscr{T}} \cup P, E_{\mathscr{T}})$, where

$$X_{\mathscr{T}} = \{x_1, \ldots, x_{|\mathscr{T}|}\}$$

and

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

A *task* t is a pair (x, w), where $x \in X_{\mathscr{T}}$ and w is the positive integer weight of t. The set of processors to which t can be *assigned* is $P_t = \{p \mid (x, p) \in E_{\mathscr{T}}\}$, that is, the set of nodes of $G_{P,\mathscr{T}}(X_{\mathscr{T}} \cup P, E_{\mathscr{T}})$ that are adjacent to x; for the sake of brevity, in the following we will always omit the subscripts '$P, \mathscr{T}$' and '$\mathscr{T}$' since the set of processors and the set of task types will be clear from the context. We will distinguish between the *unitary weight* case in which all tasks have weight 1 and the *arbitrarily weighted* case in which the weights may vary from task to task.

An *instance* σ of the on-line load balancing problem with processors P and task types $\mathscr{T}$ is then defined as a sequence of $\mathtt{new}(\cdot,\cdot)$ and $\mathtt{del}(\cdot)$ commands. In particular:

- $\mathtt{new}(x, w)$ means that a new task of weight w and type $x \in \mathscr{T}$ is created.
- $\mathtt{del}(i)$ means that the task created by the ith $\mathtt{new}(\cdot,\cdot)$ command of the instance is deleted.

Given an instance σ, a *configuration* is an assignment of the tasks of σ to the processors in P, such that each task 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 $\mathscr{A}$, let $C_h^{\mathscr{A}}$ be the configuration reached by $\mathscr{A}$ after having processed the first h commands. Moreover, let C_h^{off} be the configuration reached by the optimal off-line algorithm after having processed the first h commands. Let also set $\mathsf{opt}(\sigma) = \max_{1 \leqslant h \leqslant n} l(C_h^{\text{off}})$ and $l_{\mathscr{A}}(\sigma) = \max_{1 \leqslant h \leqslant n} l(C_h^{\mathscr{A}})$.

An on-line algorithm $\mathscr{A}$ is said to be *at most r-competitive* if there exists a constant b such that, for any instance σ, it holds that

$$l_{\mathscr{A}}(\sigma) \leqslant r \cdot \mathsf{opt}(\sigma) + b.$$

An on-line algorithm $\mathscr{A}$ is said to be *at least r-competitive* if, for any $r' < r$ and for any constant b, there exists an instance σ such that $l_{\mathscr{A}}(\sigma) > r' \cdot \mathsf{opt}(\sigma) + b$. Finally, an on-line algorithm is said to be *r-competitive* if it is both at most and at least r-competitive. We will also say that $\mathscr{A}$ has competitive ratio (at least/at most) r if $\mathscr{A}$ is (at least/at most) r-competitive.

A simple on-line algorithm for the above described load-balancing problem 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 $\mathtt{new}(x,w)$ 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 minimum and assigns the new task $t=(x,w)$ to p_i. We will analyze the behavior of this algorithm in the following sections.

2.2. The channel assignment problem formulation

Given a cellular system, that is, a set of cells positioned according to Fig. 1, we can derive the corresponding set of processors and the corresponding set of task types as follows: there is one processor for each cell and there is one task type for each simple closed curve; referring to the simplified graphical representation, there is one task type for each hexagon, one task type for each edge shared by two hexagons, and one task type for each vertex shared by three hexagons. More formally, the associated bipartite graph $G(X \cup P, E)$ is defined as follows: P denotes the set of base stations and X is defined as

$$
\begin{aligned}
X = \{x_A \mid A \text{ is a base station}\} \\
\cup \{x_{AB} \mid A \text{ and } B \text{ are two base stations that intersect each other}\} \\
\cup \{x_{ABC} \mid A, B, \text{ and } C \text{ are three base stations that intersect each other}\}.
\end{aligned}
$$

The set E is given by

$$
\begin{aligned}
E = \{(x_A, A) \mid x_A \in X\} \\
\cup \{(x_{AB}, y) \mid x_{AB} \in X \text{ and } y \in \{A, B\}\} \\
\cup \{(x_{ABC}, y) \mid x_{ABC} \in X \text{ and } y \in \{A, B, C\}\}.
\end{aligned}
$$

For example, the associated bipartite graph corresponding to the cells A, B, C, and D of Fig. 1 is shown in Fig. 2.

Each communication request $r=(p,b)$ corresponds to a task whose type $x(p)$ depends on the position p of the request and whose weight is equal to the request's bandwidth b. Hence, whenever a new communication request $r=(p,b)$ arises, a $\mathtt{new}(x(p),b)$ command is appended to the instance sequence. Moreover, if the communication request r created by the ith $\mathtt{new}(\cdot,\cdot)$ command crosses the border of a cell, then the two commands $\mathtt{del}(i)$ and $\mathtt{new}(x(p'),b)$ are appended to the instance sequence, where p' denotes the new point occupied by r.

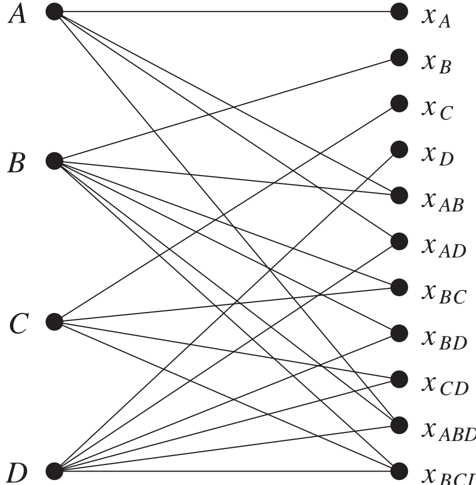


Fig. 2. The bipartite graph corresponding to cells A, B, C, and D of Fig. 1.

Clearly, the maximum cell load in a configuration of the channel assignment problem is equal to the maximum processor load in the corresponding configuration of the load-balancing problem. Hence, any result regarding the latter problem may be applied to the former one.

3. The cluster algorithm

In this section, we propose an algorithm template, called the *cluster algorithm*, to solve the load-balancing problem defined in the previous section. The basic idea of this algorithm essentially consists of suitably limiting the number of processors to which a task can be assigned. As we will see, such a limitation may result in an on-line algorithm better than the greedy one, which instead takes into account all of the available processors.

The algorithm will be introduced by referring to the bipartite graph $G(X \cup P, E)$, associated with a set of processors P and a set of task types $\mathscr{T}$.

Definition 3.1 (Cluster). Let $G(X \cup P, E)$ be a bipartite graph and let $X' \subseteq X$ and $P' \subseteq P$. Then, $C = (X', P')$ is a *cluster for* G if the graph G' induced by $X' \cup P'$ is a complete bipartite graph. We denote the sets X' and P' as $X(C)$ and $P(C)$, respectively.

Definition 3.2 (Neighborhood of cluster). Let C be a cluster for a bipartite graph G. The *neighborhood of* C, denoted by $N(C)$, is defined as the set of nodes adjacent to some node in $X(C)$, that is,

$$N(C) = \{p \in P \mid \text{there exists } x \in X(C) \text{ such that } (x, p) \in E\}.$$

Definition 3.3 (Decomposition into clusters). A set $\mathscr{S}$ of clusters for a bipartite graph $G(X \cup P, E)$ is a *decomposition into clusters of* G if every vertex in X belongs to exactly one cluster in $\mathscr{S}$ and every vertex in P belongs to at most one cluster in $\mathscr{S}$.

For example, a possible decomposition into clusters of the graph shown in Fig. 2 consists of four clusters: the first cluster includes A, x_A, and x_{AB}; the second cluster includes B and x_B; the third cluster includes C, x_C, and x_{BC}; the last cluster includes D, x_D, x_{AD}, x_{BD}, x_{CD}, x_{ABD}, and x_{BCD}.

Essentially, our algorithm consists of applying the greedy approach to a decomposition into cluster of the associated bipartite graph. More formally, given a decomposition into clusters $\mathscr{S}$ of $G(X \cup P, E)$ and given a node $x \in X$, let C_x be the cluster of $\mathscr{S}$ containing x. The *cluster algorithm* is an on-line algorithm that assigns tasks according to $\mathscr{S}$ as follows: Given a task $t = (x, w)$, the algorithm assigns t to the least loaded processor of cluster C_x.

The competitive ratio of the cluster algorithm clearly depends on the partition $\mathscr{S}$. In particular, for any $C \in \mathscr{S}$, let us consider the ratio $|N(C)|/|P(C)|$. As it can be seen, $|P(C)|$ denotes the number of processors that the cluster algorithm takes into account while assigning a task whose type belongs to C, while $|N(C)|$ is an upper bound on the number of processors that any algorithm can consider while assigning the same task. Given a decomposition $\mathscr{S}$, we define

$$r_{\mathrm{U}}^{\mathscr{S}} = \max_{C \in \mathscr{S}} \left\{ \frac{|N(C)|}{|P(C)|} \right\}$$

and

$$r_{\mathrm{W}}^{\mathscr{S}} = \max_{C \in \mathscr{S}} \left\{ \frac{|N(C)| - 1}{|P(C)|} \right\}$$

(in the following, we will always omit the superscript '$\mathscr{S}$' since the decomposition will be clear from the context).

As the following result states, the competitive ratio of the cluster algorithm in the case of unitary weights and in the case of arbitrarily weighted tasks depends on r_{U} and r_{W}, respectively.

Theorem 3.4. *For any set of processors P and any set of task types $\mathscr{T}$ and for any decomposition into clusters $\mathscr{S}$ of the associated bipartite graph, the cluster algorithm is r_{U}- and $(1 + r_{\mathrm{W}})$-competitive in the case of unitary weights and in the case of arbitrarily weighted tasks, respectively.*

Proof. Let p_i be a processor that, during the execution of the cluster algorithm, reaches the highest load $l(i)$ and let C_j be the unique cluster containing p_i. Let us consider an iteration of the cluster algorithm in which a task t is assigned to p_i so that p_i reaches a load equal to $l(i)$. Also let w be the weight of task t. Since t is assigned to p_i whose load, before the arrival of t, is $l(i) - w$, we have that each processor in $P(C_j)$ has load at least $l(i) - w$. This implies that the overall weight of the tasks generated within C_j, after the arrival of t, is at least $|P(C_j)|(l(i) - w) + w$. Notice that, from

Definition 3.1, $N(C_i)$ is the set of processors such tasks can be (off-line) assigned to. Hence, if l^* denotes measure of an optimal off-line solution, then it holds that

$$(l(i) - w) \cdot |P(C_j)| + w \leqslant |N(C_j)| \cdot l^*,$$

thus implying

$$\frac{l(i)}{l^*} = \frac{l(i) - w}{l^*} + \frac{w}{l^*} \leqslant \frac{|N(C_j)|}{|P(C_j)|} + \frac{w}{l^*}\left(1 - \frac{1}{|P(C_j)|}\right).$$

This proves the upper bound on the competitiveness of the cluster algorithm both in the unitary weights and in the arbitrarily weighted case: notice that, in the former case, we use the fact that all tasks have weight 1, while, in the latter case, we use the fact that $w \leqslant l^*$.

We now show that our analysis is tight. As for the unitary weight case, we observe that, for any positive integer l^*, it is possible to generate $|N(C_j)| \cdot l^*$ tasks in the positions included in $X(C_j)$ so that these tasks can be assigned by an optimal off-line solution to the set $N(C_j)$ without overcoming the load l^*. By definition, the cluster algorithm will assign these tasks to the processor set $P(C_j)$, so that at least one among these processors will reach a load greater than or equal to $|N(C_j)|/|P(C_j)|l^*$. As for the case of arbitrarily weighted tasks, once again we observe that, for any positive integer l^*, it is possible to generate $(|N(C_j)| - 1)l^*$ tasks of weight 1 in the positions included in $X(C_j)$ so that these tasks can be assigned by an optimal off-line solution to the set $N(C_j)$ without overcoming the load l^* and without assigning any task to a specific processor $p^* \in N(C_j)$. By definition, the cluster algorithm will assign these tasks to the processor set $P(C_j)$, so that *every* processor in $P(C_j)$ will reach a load greater than or equal to $(|N(C_j)| - 1)l^*/|P(C_j)| - 1$. Hence, if we now generate a task of weight l^* in a position which can be served by p^*, then at least one processor in $P(C_j)$ will reach a load greater than or equal to $(|N(C_j)| - 1)l^*/|P(C_j)| - 1 + l^*$, while the optimal off-line solution can assign all the tasks without overcoming the load l^*. Hence the theorem follows. □

Example 3.5. Consider the case of the on-line load balancing problem on identical machines with no restrictions, i.e., each task can be assigned to any processor in $P = \{p_1, \ldots, p_n\}$. We can easily represent this problem as a bipartite graph $G(X \cup P, E)$, where $X = \{x\}$ and $(x, p_i) \in E$, for $1 \leqslant i \leqslant n$. In this case, it is easy to see that the best decomposition into clusters of G is the one formed by only one cluster, that is, the graph itself. In this case, the cluster algorithm reduces to the greedy algorithm proposed by Graham [14,15]. Indeed, Theorem 3.4 implies that such an algorithm is $(2 - 1/n)$-competitive in the case of arbitrarily weighted tasks, which is optimal [4]. (Notice that, in the case of permanent tasks, there exists an algorithm with competitive ratio strictly less than 2, see e.g. [1].)

As a consequence of the above result, it follows that, in order to obtain a good competitive ratio for the cluster algorithm, we have to choose a decomposition into clusters that maximizes the ratio between the number of processors of any cluster and the size of its neighborhood.

4. Application to hexagonal grid topology

We now apply the cluster algorithm to the channel assignment problem: to achieve this goal, in this section we will always implicitly refer to the bipartite graph associated with the cellular network (or to a finite portion of it). In order to prove our main theorem, in the next lemma we explicitly give a decomposition of this graph into clusters.

Lemma 4.1. *There exists a decomposition* $\mathscr{S}$ *into clusters such that, for any cluster* $C \in \mathscr{S}$, $|P(C)| = 1$ *and* $|N(C)| = 4$.

Proof. Let us consider a cell D and let E, F, G, C, B, and A be its neighbor cells in clockwise order (see Fig. 1). The cluster containing D is defined as

$$(\{x_D, x_{AD}, x_{BD}, x_{CD}, x_{ABD}, x_{BCD}\}, \{D\}).$$

Hence, the neighborhood of the cluster is $\{A, B, C, D\}$ (see also Fig. 2). By considering a cluster for each cell, we can easily obtain a decomposition into clusters: Hence, the lemma follows. □

As a consequence of the above lemma and of Theorem 3.4, we have the following result.

Theorem 4.2. *The cluster algorithm is* 4-*competitive both in the case of unitary weights and in the case of arbitrarily weighted tasks.*

It is worth observing that the cluster decomposition described above yields a "fixed" allocation scheme in which every region of the plane is assigned to one cell. Therefore, no communication is needed to decide which cell of a cluster is currently the least loaded.

We now prove that the competitive ratio of the greedy algorithm is strictly worse than the competitive ratio of the cluster algorithm. In the following, we will denote by $\texttt{new}(x,w)^l$ the sequence $\texttt{new}(x,w)\cdots\texttt{new}(x,w)$ of length l.

Theorem 4.3. *The greedy algorithm is at least* 5-*competitive, even in the case of unitary weights.*

Proof. We show a unitary weight instance σ that has optimal cost 1, while the greedy algorithm computes an assignment of cost 5. Observe that, by cloning the commands, this is sufficient to prove that the greedy algorithm is at least 5-competitive, even in the case of unitary weights. The instance refers to the topology shown in Fig. 3 and assumes that the greedy algorithm solves ties by selecting the alphabetically greatest cell. Clearly, the instance can be adapted to any different criterion.

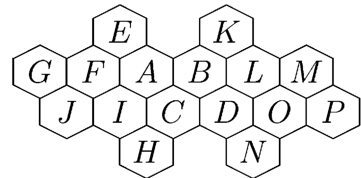


Fig. 3. The cellular network of the proof of Theorem 4.3.

The instance consists of two parts. The first part is formed by the following commands:

- $\mathtt{new}(x_{AE},1)^2\mathtt{new}(x_{FG},1)^2\mathtt{del}(1)\mathtt{del}(3)\mathtt{new}(x_{AF},1)^2$.
- $\mathtt{new}(x_{BK},1)^2\mathtt{new}(x_{LM},1)^2\mathtt{del}(7)\mathtt{del}(9)\mathtt{new}(x_{BL},1)^2$.
- $\mathtt{new}(x_{CH},1)^2\mathtt{new}(x_{IJ},1)^2\mathtt{del}(13)\mathtt{del}(15)\mathtt{new}(x_{CI},1)^2$.
- $\mathtt{new}(x_{DN},1)^2\mathtt{new}(x_{OP},1)^2\mathtt{del}(19)\mathtt{del}(21)\mathtt{new}(x_{DO},1)^2$.
- $\mathtt{del}(5)\mathtt{del}(11)\mathtt{del}(17)\mathtt{del}(23)$.

It is easy to see that after processing this first sequence of commands, the greedy algorithm reaches a configuration in which each of the four cells A, B, C, and D has load 2, while in the optimal off-line solution the four cells have load 0.

The second part of the instance is formed by the following commands:

- $\mathtt{new}(x_{AB},1)^2\mathtt{del}(26)\mathtt{new}(x_{BD},1)\mathtt{new}(x_{CD},1)$.
- $\mathtt{del}(27)\mathtt{new}(x_{BC},1)^2\mathtt{del}(29)\mathtt{new}(x_{B},1)$.

It is easy to see that after processing this second part, the greedy algorithm reaches a configuration in which cell B has load 5, while in the optimal off-line solution all cells have load 1. Hence, the theorem is proved. □

Finally, the following lower bound holds for any on-line algorithm applied to the channel assignment problem in cellular networks.

Theorem 4.4. *Any on-line algorithm for the channel assignment problem in cellular networks is at least* 3*-competitive, even in the case of unitary weights.*

Proof. Consider the cellular network in Fig. 3. In particular, we will use only the five cells A, B, C, D, and E. In order to prove the lower bound, we now show an input sequence σ such that: (a) any on-line algorithm with input σ reaches a *critical* configuration, that is, a configuration in which a cell X exists whose load is equal to 2, and (b) there exists an optimal off-line solution for σ of measure 1 that reaches a configuration in which X is empty.

The sequence σ starts with $\mathtt{new}(x_{AE},1)$: Without loss of generality, we may assume that the on-line algorithm assigns this task to cell A. Let us now consider the following extension of σ:

- $\mathtt{new}(x_{AB},1)\mathtt{new}(x_{BD},1)\mathtt{new}(x_{CD},1)$.

Observe that any on-line algorithm that does not assign one of these new three tasks according to the greedy criterion reaches a critical configuration in which the role of X is played by A, B, and D, respectively. Hence, at the end of the above sequence of operations, A, B, C, and D have been assigned tasks 1, 2, 4, and 3, respectively. At this point, we first delete task 3 and then create the following new task:

- $\mathtt{new}(x_{BC}, 1)$.

If this task is assigned to B, then the on-line algorithm has reached a critical configuration, in which the role of X is played by B. Otherwise, the task is assigned to C, and the on-line algorithm has reached a critical configuration, in which the role of X is played by C.

Once the on-line algorithm has reached a critical configuration, we create a new task within cell X. It is then easy to see that: (a) the configuration reached by the on-line algorithm has load greater than or equal to 3, while (b) there exists an optimal off-line solution that reaches a configuration whose load is 1. This immediately implies the theorem in the case of arbitrarily weighted tasks: indeed, it suffices to send tasks of weight l sufficiently large, instead of tasks with weight 1. Moreover, it is possible to modify the above instance in order to extend the result to the case of unitary weights. To achieve this goal, it suffices to clone each $\mathtt{new}(\cdot,\cdot)$ command of the sequence (except the last one) into $2l$ commands. Notice that, for each original command, any algorithm must assign (at least) l clones to one of the two available cells (let us say X). By inserting an appropriate sequence of at most l $\mathtt{del}(\cdot)$ operations between two sequences of $\mathtt{new}(\cdot,\cdot)$ commands, we can remove the tasks not assigned to X. Moreover, every $\mathtt{del}(\cdot)$ operation of the original instance is also cloned into an appropriate sequence of l commands. Finally, the last $\mathtt{new}(\cdot,\cdot)$ command of the original instance is cloned into l commands. This sequence of unitary tasks essentially mimics the one with tasks of weight l. Hence, the theorem follows. □

5. Application to linear topologies

An interesting case of on-line load balancing with restricted assignment is the case in which processors are positioned on a line at unitary distance, the set of positions in which the tasks can arise coincide with the set of positions of the processors, and each task can be assigned to processors at distance at most k from the processor the task arose in, i.e., processor p_i can serve tasks arising in the interval $\{i-k,\dots,i+k\}$. This situation is another special case of the general load-balancing problem which has been already studied: indeed, it models, for example, the case of radio networks where all stations have the same transmission range [18,21], and that of an array of processors where the communication cost between two processors depends on their distance [13] (see also [7,17] for other one-dimensional variations of the on-line load-balancing problem). In Table 2, we summarize the results in the case of *arbitrarily weighted tasks*. Particular attention should be given to the case $k=1$ which is strictly related to our original motivating problem. Indeed, in [8] one-dimensional frequency

Table 2
Competitiveness results for linear topologies and weighted tasks

	Greedy	Our algorithm	Lower bound
$k = 1$	At least 3 (Theorem 5.4)	$\frac{5}{2}$ (Corollary 5.3)	$\frac{5}{2}$ (Theorem 5.8)
$k > 1$	At least $\frac{8k+1}{2k+1}$ (Theorem 5.5)	$\frac{4k+1}{k+1}$ (Corollary 5.3)	$\frac{3k+1}{k+1}$ (Theorem 5.9)

assignment problems have been investigated. This restriction of the problem is motivated by vehicular technology applications, where the users are located on highways (represented by a line). Notice that, when cellular systems with bigger overlapping regions occur, such as in the model proposed in [19], the corresponding network topology is our one-dimensional model with $k = 1$. For this case, we can show that, when arbitrarily weighted tasks occur, (a) the greedy algorithm is 3-competitive: observe that this performance ratio is guaranteed by any trivial algorithm which "blindly" assigns tasks to a fixed processor among the three available; and (b) the cluster algorithm is 2.5-competitive, which matches the lower bound (see Table 2).

The associated bipartite graph is defined as $G(X \cup P, E)$, where $X = \{x_0, \ldots, x_n\}$, $P = \{p_0, \ldots, p_n\}$, and there exists an edge (x_i, p_j) if and only if $|i - j| \leqslant k$. In this section, we will always implicitly refer to this graph.

Let $\mathscr{A}_{\text{triv}}$ be the *trivial* algorithm which assigns each task to the processor the task has been created in. The following fact easily follows from the observation that the set of tasks assigned by $\mathscr{A}_{\text{triv}}$ to a processor could be shared in the optimal off-line solution among at most $2k + 1$ processors.

Fact 1. *For all k, $\mathscr{A}_{\text{triv}}$ is at most $2k + 1$-competitive.*

5.1. Upper bounds

Let us describe how the bipartite graph can be partitioned into clusters so that the cluster algorithm performs better than the greedy one. Basically, the set of positions will be decomposed into consecutive disjoint intervals of size $k + 1$: however, if n is not a multiple of $k + 1$, then the first and the last intervals have to be appropriately defined in order to deal with the "border" processors.

Formally, let $n_c = n \operatorname{div}(k + 1)$ and let $r = n \bmod (k + 1)$. Notice that, without loss of generality, we may assume that $r > 0$. If $r \geqslant (k + 1)/2$, then we set the cardinality c_0 of the first cluster equal to $k + 1$, otherwise we set $c_0 = \lceil (r + k + 1)/2 \rceil$. Now, for any position i in $\{0, c_0, c_0 + (k + 1), c_0 + 2(k + 1), \ldots, c_0 + (n_c - 1)(k + 1)\}$, we define the cluster C_i as follows:

$$X(C_i) = \begin{cases} \{x_0, x_1, \ldots, x_{c_0 - 1}\} & \text{if } i = 0, \\ \{x_i, x_{i+1}, \ldots, x_{\min(i+k,n)}\} & \text{otherwise} \end{cases}$$

and

$$P(C_i) = \begin{cases} \{p_0, p_1, \ldots, p_{c_0-1}\} & \text{if } i = 0, \\ \{p_i, p_{i+1}, \ldots, p_{\min(i+k,n)}\} & \text{otherwise.} \end{cases}$$

It is easy to see that the sets C_i satisfy Definitions 3.1 and 3.3. The next theorem states an upper bound on the ratio between the number of processors that any algorithm can consider while assigning a new task and the number of processors that the cluster algorithm takes into account.

Theorem 5.1. *For any cluster* C,

$$\frac{|N(C)|}{|P(C)|} \leqslant \frac{3k+1}{k+1}.$$

Proof. Observe that, for any cluster C, either $|P(C)| = k+1$ and $|N(C)| \leqslant 3k+1$ or $|N(C)| \leqslant |P(C)| + k$. In order to prove the theorem, it then suffices to show that $|P(C)| \geqslant (k+1)/2$, for any cluster C. To achieve this goal, recall that $r = n \bmod (k+1)$ and that if $r \geqslant (k+1)/2$, then $|P(C)| \geqslant (k+1)/2$, for any cluster C. Otherwise (that is, $r < (k+1)/2$), for any cluster C,

$$|P(C)| \geqslant \left\lfloor \frac{r+k+1}{2} \right\rfloor \geqslant \left\lfloor \frac{k+2}{2} \right\rfloor = \left\lfloor \frac{k}{2} \right\rfloor + 1 \geqslant \frac{k-1}{2} + 1 = \frac{k+1}{2},$$

where the second inequality is due to the fact that we assumed $r \geqslant 1$. Hence, the theorem follows. □

From the above theorem and from Theorem 3.4, we obtain the following two results.

Corollary 5.2. *The cluster algorithm is* $(3k+1)/(k+1)$*-competitive in the case of unitary weights.*

Corollary 5.3. *The cluster algorithm is* $(4k+1)/(k+1)$*-competitive in the case of arbitrarily weighted tasks.*

It is worth observing that in the above cluster decomposition, each cluster contains $k+1$ processors. Therefore, in order to decide where to assign a new task, the cluster algorithm needs to query only $k+1$ processors, while the greedy one queries all $2k+1$ available processors.

In the rest of this section we will prove that the cluster algorithm has a competitive ratio smaller than the greedy one, while in Section 5.2 we will show that its competitive ratio is optimum for $k = 1$.

5.1.1. The greedy algorithm

In the sequel of the paper, we will refer to a configuration C as a sequence $c_1 \cdots c_m$ of 4-tuples such that the hth tuple $c_h = \langle t_h, i_h, j_h, s_h \rangle$ specifies that the task with index t_h

Table 3
The greedy worst-case instance

Instance	Greedy configuration	Off-line configuration
$\text{new}(1,1)^3$	1 2 3 0 **1** 2 3 4 5	3 1 2 0 **1** 2 3 4 5
$\text{new}(2,1)^3$	6 4 3 5 0 1 **2** 3 4 5	3 5 4 6 0 1 **2** 3 4 5
$\text{new}(-2,1)^3$	6 7 8 9 4 -3 **-2** -1 0 1 **2**	9 7 8 4 6 -3 **-2** -1 0 1 2 3
$\text{new}(-1,1)$	6 8 9 10 4 -3 -2 **-1** 0 1 **2**	9 10 8 4 6 -3 -2 **-1** 0 1 2 3
$\text{new}(0,1)^3$	13 11 12 6 9 10 4 -3 -2 -1 **0** 1 2	9 10 11 12 13 4 6 -3 -2 -1 **0** 1 2 3

has been created in processor p_{i_h}, has weight j_h, and is currently assigned to processor $p_{i_h+s_h}$ where $s_h \in \{-k,\dots,0,\dots,k\}$. Moreover, we will identify two configurations which are equivalent modulo a re-indexing of their tasks and of the processors. For the sake of clarity, we will usually represent a configuration in a graphical way: for instance, the configuration

$$\langle 1,i,1,0\rangle\langle 2,i,1,1\rangle\langle 3,i+3,1,-1\rangle$$

is graphically represented as shown in the middle column of the first row of Table 10; notice that we have specified the tasks' indices "1", "2" and "3" only for the sake of clarity. Observe that such a graphical representation specifies neither the origin nor the weight of a task, since this information will be anyway specified by the accompanying text. Finally, given a configuration C and a sequence of `del(·)` commands, we will graphically represent the application of this sequence to C by coloring the rectangles associated with the deleted tasks with the gray color (see, for example, the middle column of the first row of Table 3 where

$$C = \langle 1,1,1,-1\rangle, \langle 2,1,1,0,\rangle, \langle 3,1,1,1\rangle$$

and the sequence of delete commands is `del(1)del(2)`).

The next result states that the greedy algorithm does not perform better than the trivial algorithm $\mathscr{A}_{\text{triv}}$ (see Fact 1), when applied to the case $k=1$. Observe that, since we are dealing with arbitrarily weighted tasks, it suffices to show an instance with optimal off-line equal to 1 and such that the greedy algorithm reaches a configuration with measure 3.

Theorem 5.4. *For $k=1$, the greedy algorithm is at least 3-competitive.*

Proof. Let us consider the following instance:

$$\texttt{new}(1,1)^3\texttt{del}(1)\texttt{del}(2)\texttt{new}(2,1)^3\texttt{del}(3)\texttt{del}(5)$$

$$\texttt{new}(-2,1)^3\texttt{del}(7)\texttt{new}(-1,1)\texttt{del}(8)\texttt{new}(0,1)^3.$$

Assume that ties are solved in a left-to-right order. The proof does not really depend on this assumption, and can be easily generalized to variants of the greedy algorithm that solve ties by assigning a task to *any one of the less loaded available processors*. The behavior of the greedy algorithm with input the above instance is the one shown in the middle column of Table 3: the right column, instead, shows an optimal off-line solution.

As shown in the table, the greedy algorithm will assign to processor p_{-1} three tasks while the off-line optimum solution will be able to assign the new three tasks to the empty processors p_{-1}, p_0, and p_1. In other words, the load of the final configuration provided by the greedy algorithm is 3 while the optimal off-line solution has load 1. Hence, the theorem follows. □

Let us now prove a lower bound on the competitive ratio of the greedy algorithm in the case of arbitrarily weighted tasks and $k \geqslant 2$.

Theorem 5.5. *For any* $k > 1$, *the greedy algorithm in the case of arbitrarily weighted tasks is at least* $(8k+1)/(2k+1)$*-competitive.*

Proof. For the sake of clarity, we first prove the theorem for the case $k=2$. The proof for any $k > 2$ will be then given as a generalization of the previous case. □

5.1.1.1. The case $k=2$. Similarly to the proof of Theorem 5.4, we describe an instance σ such that the greedy algorithm on input σ is 17/5-competitive.[4] For the sake of clarity, we split σ into three parts, that is $\sigma=\sigma^{\mathrm{l}}\sigma^{\mathrm{r}}\bar{\sigma}$. In Table 4 we describe σ^{l} and the behavior of the greedy algorithm; for the sake of brevity, we do not explicitly show the delete commands which are, instead, graphically indicated. The aim of the subsequence $\sigma^{\mathrm{l}}\sigma^{\mathrm{r}}$ is to "force" the greedy algorithm to reach the configuration $\bar{C}$ shown in the first row of Table 6. In particular, the "left" part of configuration $\bar{C}$ (i.e., tasks assigned by the greedy algorithm to processors p_7, p_8 and p_9) is due to σ^{l} (see the last row of Table 4). Similarly, we can obtain the "right" part of configuration $\bar{C}$ (i.e., tasks assigned by the greedy algorithm to processors p_{10}, p_{11} and p_{12}) from the sequence σ^{r}_k (see Table 5).

In Table 6 we show the behavior of the greedy algorithm starting from $\bar{C}$ on input $\bar{\sigma}$. It is easy to see that the measure of the greedy algorithm is 17, while an optimum off-line assignment of value 5 exists.

5.1.1.2. The general case. The proof for any $k > 2$ is a generalization of the proof given for $k=2$: we generalize the instance σ and the configuration $\bar{C}$ used above to

[4] Again, we assume ties are broken left-to-right. It is possible to see that different tie break rules can be managed by slightly modifying the instance in order to obtain the same result.

Table 4
The proof of Theorem 5.5 for $k=2$ (the sequence σ^{l})

Instance	Greedy configuration	Off-line configuration
$\mathrm{new}(2,5)^5$		
$\mathrm{new}(5,1)^6$, $\mathrm{new}(6,1)^2$, $\mathrm{new}(7,1)$, $\mathrm{new}(7,5)$, $\mathrm{new}(6,1)^3$, $\mathrm{new}(6,5)$		
$\mathrm{new}(5,5)$		

$\sigma_k = \sigma_k^{\mathrm{l}}\sigma_k^{\mathrm{r}}\overline{\sigma_k}$ and $\overline{C_k}$, respectively. In particular, Tables 7 and 8 show the configuration $\overline{C_k}$ and the execution of the greedy algorithm, starting from such configuration, on input $\overline{\sigma_k}$. Notice that, tasks assigned to processors in $[2k,3k]$ (respectively, $[3k+1,4k+1]$) arose in the interval $[k,2k]$ (respectively, $[4k+1,5k+1]$). Additionally, such tasks can be (off-line) assigned to the processors in the intervals $[0,2k-1]$ and $[4k+1,6k+1]$, respectively (see the configuration $\bar{C}_k^{\mathrm{off}}$ in Table 9). Also notice that, by setting $h=2k-2$ and $l^*=2k+1$, the greedy algorithm yields a solution of cost $3l^*+h=8k+1$ (see Table 8), while the optimum off-line is $l^*=2k+1$ (see Table 9). So, in order to prove the theorem it remains to describe the sequence $\sigma_k^{\mathrm{l}}\sigma_k^{\mathrm{r}}$ which forces the greedy algorithm to reach the configuration $\overline{C_k}$, while an assignment to those tasks exists so to obtain the configuration $\overline{C_k}^{\mathrm{off}}$.

To achieve this goal, we will denote by $last_i$ the index of the last task assigned (by the greedy algorithm) to processor in position i. We first consider the instance σ^{l} in Table 4 and we generalize it as follows:

- $\texttt{new}(k,l^*)^{2k+1}\underbrace{\texttt{del}(last_0)\cdots\texttt{del}(last_{k-1})}_{k}$
- $\texttt{new}(2k+1,1)^{(k+1)h}\underbrace{\texttt{new}(2k+2,1)^h\texttt{new}(2k+3,1)^h\cdots\texttt{new}(3k,1)^h}_{k-1}$

Table 5
The proof of Theorem 5.5 for $k=2$ (the sequence σ^r)

Instance	Greedy configuration	Off-line configuration
$\mathrm{new}(17,5)^5$		
$\mathrm{new}(14,1)^6$, $\mathrm{new}(13,1)^2$, $\mathrm{new}(12,1)^2$,		
$\mathrm{new}(14,5)$, $\mathrm{new}(13,1)^3$,		
$\mathrm{new}(13,5)$, $\mathrm{new}(12,5)$		

- $\mathtt{new}(3k+1,1)^{h/2}\mathtt{new}(3k+1,l^*)$
- $\mathtt{new}(3k,1)^{2k}\mathtt{del}(last_{4k})$
- **step** 1: $\mathtt{new}(3k,l^*)\mathtt{del}(last_{4k-1})$
  step 2: $\mathtt{new}(3k-1,l^*)\mathtt{del}(last_{4k-2})$
  ⋮
  step k: $\mathtt{new}(2k+1,l^*)\mathtt{del}(last_{3k})$
- delete all the tasks assigned to processors in $[k,3k]$.

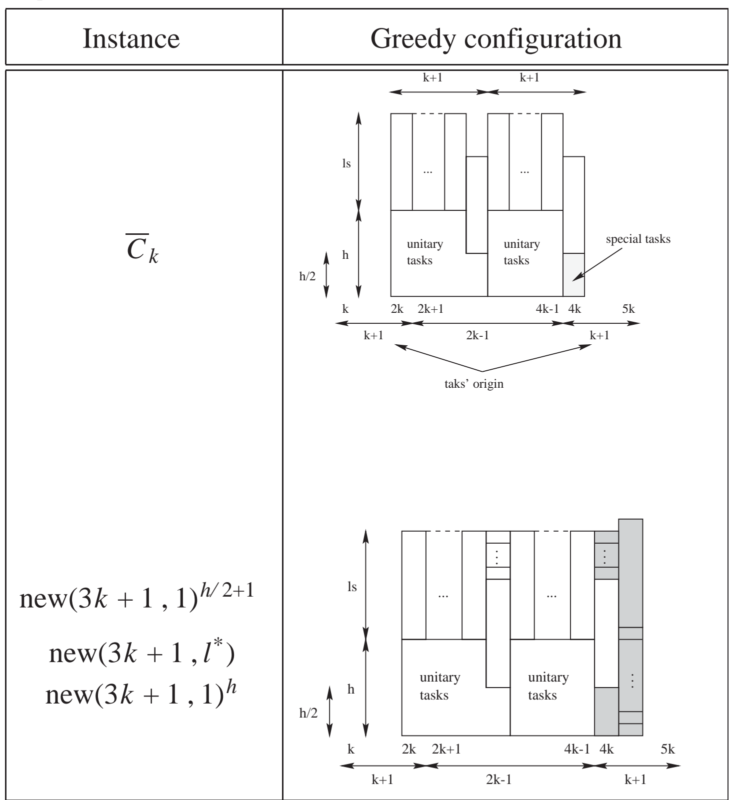
It is worth observing that an (off-line) assignment to the above sequence exists such that: (a) all the tasks of weight 1 can be assigned to processors of index $2k+2$ or greater (in particular, the $k\cdot h+h/2$ *surviving* ones can be assigned to $[2k+2,3k]$); (b) each task of weight l^* can be assigned to a processor in the interval $[0,2k+1]$:

Table 6
The proof of Theorem 5.5 for $k=2$ (the configuration $\bar{C}$ and the sequence $\bar{\sigma}$)

Inst./Conf.	Greedy configuration	Off-line configuration
$\overline{C}$		
$\text{new}(11,1)^3$, $\text{new}(11,3)$, $\text{new}(11,1)^2$		
$\text{new}(10,5)^2$		
$\text{new}(9,5)^5$		

the $k+1$ tasks arising in positions $[2k+1, 3k+1]$, can be assigned to the interval $[k+1, 2k+1]$; (c) at each step the load of every processor is at most l^*. On the other hand, the greedy algorithm, after processing the above sequence, yields a solution in which all the remaining tasks are assigned to the interval $[3k+1, 4k+1]$. Moreover, the load of processors in $[3k+1, 4k]$ will be $h+l^*$, while the load of the processor in $4k+1$ will be equal to $h/2+l^*$. Thus, by shifting on the left (i.e., re-indexing the tasks' origin) the above sequence by a factor $k+1$, we obtain the sequence σ_k^1 which

Table 7
The proof of Theorem 5.5 for any $k > 2$ (the configuration $\bar{C}_k$ and the sequence $\bar{\sigma}_k$, part 1)

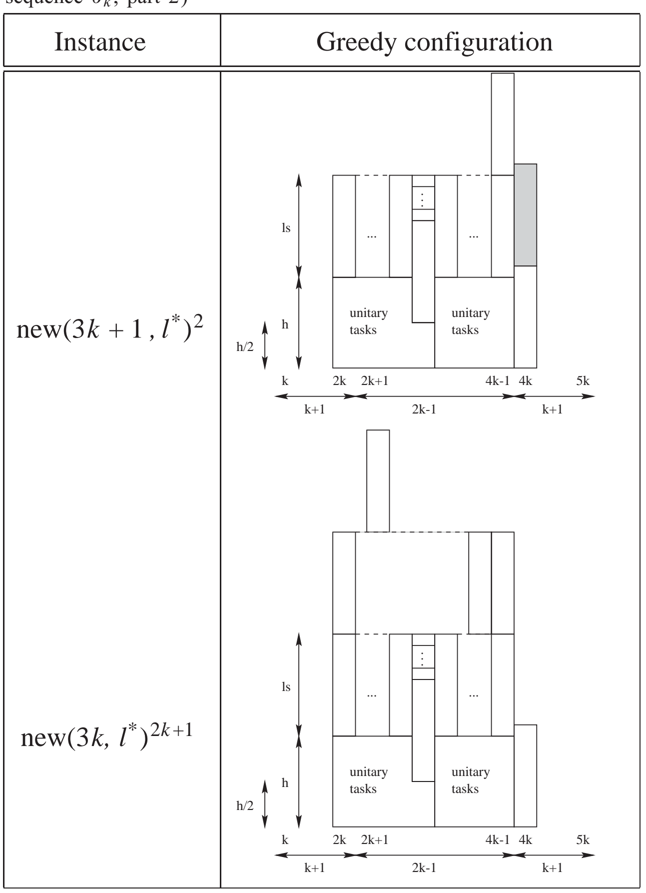


yields the "left" part of the configuration $\overline{C_k}$ (i.e., the tasks assigned by the greedy algorithm to processors in $[2k, 3k]$ in Table 9).

The sequence σ_k^{r} can be obtained by using a symmetric argument. Indeed, let us consider the following sequence:

- $\texttt{new}(0, l^*)^{2k+1}\underbrace{\texttt{del}(last_{k+2})\texttt{del}(last_{k+3})\ldots\texttt{del}(last_{2k+1})}_{k}\texttt{new}(-k-1, 1)^{h(k+1)}$
- **step** 1: $\texttt{new}(-k-2, 1)^{l^*}$
  step 2: $\texttt{new}(-k-3, 1)^{l^*}$
  $\vdots$
  step k: $\texttt{new}(-2k-1, 1)^{l^*}$
- $\texttt{new}(-2k-1, 1)^{2k+1}$
- $\underbrace{\texttt{del}(last_{-3k-1})\texttt{del}(last_{-3k})\ldots\texttt{del}(last_{-2k-1})}_{k+1}$

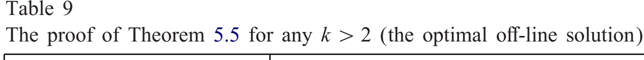
Table 8
The proof of Theorem 5.5 for any $k > 2$ (the configuration $\bar{C}_k$ and the sequence $\bar{\sigma}_k$, part 2)

Instance	Greedy configuration
$\text{new}(3k+1, l^*)^2$	
$\text{new}(3k, l^*)^{2k+1}$	

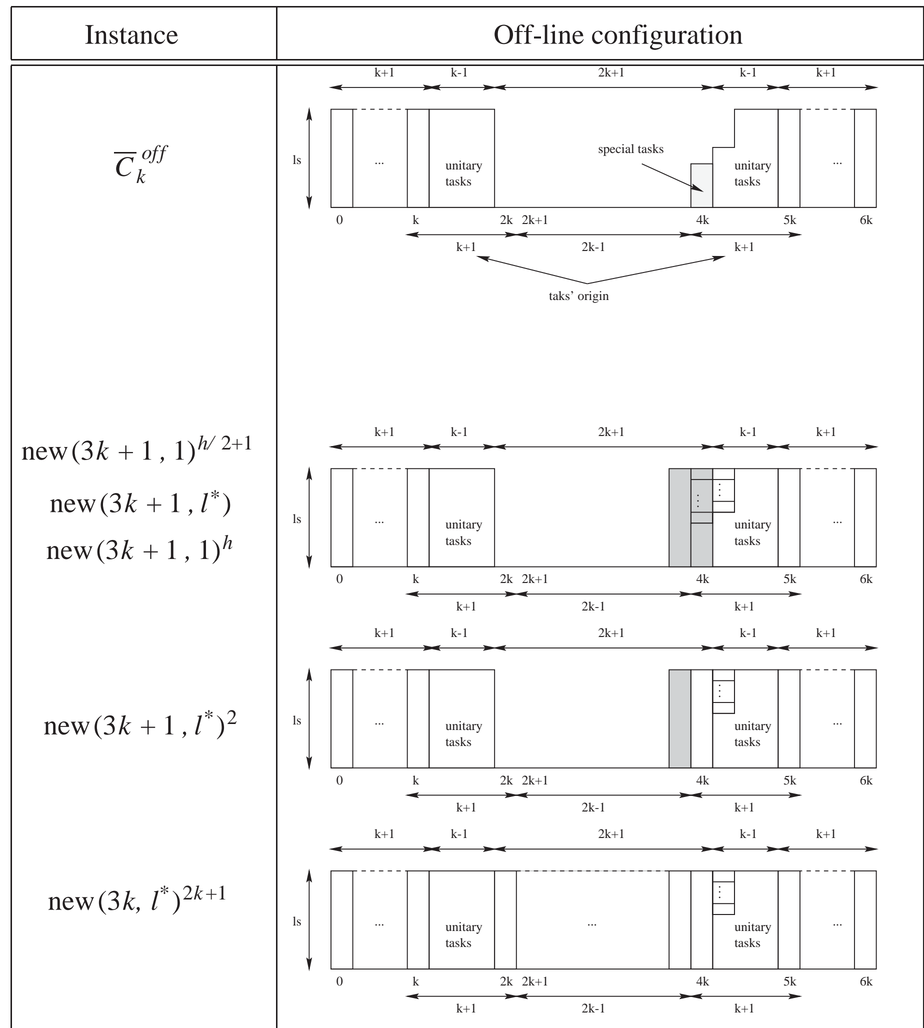
- $\texttt{del}(last_{-2k-1})^{h/2}$
- **step** 1: $\texttt{new}(-k-1, l^*)\texttt{del}(last_{k-1})^{h+1}$
  step 2: $\texttt{new}(-k-2, l^*)\texttt{del}(last_{k-2})^{h+1}$
  ⋮
  step k: $\texttt{new}(-2k, l^*)\texttt{del}(last_{-2k})^{h+1}$
  step $k+1$: $\texttt{new}(-2k-1, l^*)$
- delete all the tasks in $[-k, 0]$.

Notice that, it is possible to assign (off-line) the tasks of the above sequence in such a way that: (a) the h out of the $h(k+1)$ unitary tasks arising in position $-k-1$ that survive at the end of the sequence are all assigned to p_{-2k-1}; (b) at the generic **step** i, because of the $\texttt{del}(\cdot)$ commands, processor p_{-i} has no task and it can be used to

Table 9



The proof of Theorem 5.5 for any $k > 2$ (the optimal off-line solution)



assign the new task of weight l^* (in particular, up to **step** 1 no task is assigned to p_{-1}); (c) after the last `del(·)` command all the surviving tasks are assigned to the interval $[-2k-1, -1]$ without overcoming the load l^* (in particular, p_{-2k-1} and p_{-2k} have h and $l^* - h$ unitary tasks, respectively; in $[-2k+1, -k]$ each processor has l^* unitary tasks; in $[-k+2, -1]$ each processor has a task of weight l^*). This can be verified by considering the execution of the greedy algorithm on the above sequence, which yields the set of surviving tasks at each time step. On the other hand, the greedy algorithm applied to this sequence reaches the following configuration: In $[-3k-1, -2k-2]$ each processor has load $h + l^*$ and p_{-2k-1} has load $h/2 + l^*$.

Table 10
"Bad" configurations

Configuration	Graphical representation	Name
$1, i, 1, 0 \quad 2, i, 1, 1$ $3, i+3, 1, \quad 1$	1 2 3 i i+1 i+2 i+3 i+4 i+5	C_0
$1, i, 1, 0 \quad 2, i, 1, 1$ $3, i+d+3, 1, \quad 1 \quad 4, i+d+3, 1, 0$	1 2 ... 3 4 i i+1 d i+d+3	C_d

Hence, by shifting the whole sequence to the right by a factor $6k+2$ we obtain the sequence σ_k^{r} (see also the configurations $\overline{C_k}$ and $\overline{C_k}^{\mathrm{off}}$ in Tables 7 and 9).

5.2. Lower bounds

In this section we prove some lower bounds on the competitiveness of any on-line algorithm (see the last column of Table 2). In particular, for $k=1$, we prove that the upper bound given by the cluster algorithm cannot be improved, while, for $k>1$, we show that the cluster algorithm is not too far from being optimal. Notice that, the result in [4] implies a lower bound equal to $2-1/(2k+1)$: indeed, the case of n identical machines (see Example 3.5) can be easily simulated on the linear topology in which $2k+1=n$. In the sequel we will improve this lower bound in the case of linear topologies.

In order to prove that, for $k=1$, any on-line strategy cannot be less than 5/2-competitive (when arbitrarily weighted tasks are considered), we make use of the "bad" configurations shown in Table 10.

Lemma 5.6. *For any on-line algorithm $\mathscr{A}$, a sequence $\sigma^1_{\mathscr{A}}$ exists such that either*

$$l_{\mathscr{A}}(\sigma^1_{\mathscr{A}}) \geqslant 5 \cdot \mathsf{opt}(\sigma^1_{\mathscr{A}})/2$$

or

$$A(\sigma^1_{\mathscr{A}}) \text{ passes through } C_d \text{ for some } d \geqslant 0.$$

Proof. We will first show that either $\mathscr{A}$ is at least 5/2-competitive, or it eventually reaches a configuration in which three tasks arising in position i are assigned to p_{i-1}, p_i and p_{i+1}. Let $\overline{C^i}$ denote such configuration. Table 11 shows the first part of the sequence $\sigma^1_{\mathscr{A}}$. Notice that, either $\mathscr{A}$ reaches the configuration shown in the lowest row (or a symmetric one with two tasks assigned to p_0 and one task assigned to p_2), or it reaches the above desired configuration $\overline{C^i}$.

We can therefore assume that the configuration in the second row of Table 11 has been reached. Starting from such configuration, in Table 12 we complete the sequence $\sigma^1_{\mathscr{A}}$. Notice that, at any step a different choice would yield either a 5/2-competitive solution or the configuration $\overline{C^i}$. Indeed, the configuration $\overline{C^5}$ can be obtained from the

Table 11
The proof of Lemma 5.6: first part

Instance	On-line solution	Remark	Off-line solution
new$(1,1)^3$		Otherwise jump to 2nd part	
new$(3,1)^3$		Use p_3 and p_4	

Table 12
The proof of Lemma 5.6: second part

Instance	On-line solution	Remark	Off-line solution
new$(6,1)^3$		Otherwise C_3	
new$(4,2)^3$		Must use p_5	
new$(5,2)^3$		No other way	

last row of Table 12 by terminating all the tasks but 13, 14 and 15. Notice that, even though the surviving tasks will have weight equal to 2, this is not a problem since it suffices to rescale the subsequent tasks by a factor 2.

Finally, we observe that a "bad" configuration C_d, for some $d \geqslant 0$, can be obtained by combining any two configurations $\overline{C^i}$ and $\overline{C^{i+d+3}}$ and then by killing the two tasks assigned to processors p_{i-1} and p_{i+d+4}. However, we need these two configurations to be reached by using *tasks of the same weight* (namely, of weight 1 or of weight 2). This, in turn, can be obtained by suitably "cloning" the sequence of tasks described above into three sequences. Indeed, if after the first 9 commands of each clone (corresponding to the first row of Table 12) two out of three of such clones reached the configurations $\overline{C^i}$ and $\overline{C^{i+d+3}}$, respectively, then we simply delete all the tasks of the other clone. Otherwise, there are two out of three copies that reached the configuration of the first row of Table 12. Hence, such copies will reach two configurations equivalent to $\overline{C^i}$ and $\overline{C^{i+d+3}}$ in which *all* tasks have weight 2. □

Table 13
The proof that C_0 is a dead-end configuration

Inst./Conf.	On-line solution	Remark	Off-line solution
C_0	1 2 3 i i+1 i+2 i+3 i+4 i+5		2 1 3 i-1 i i+1 i+2 i+3 i+4 i+5
$\text{new}(i+2,2)^3$	4 5 1 2 3 6 i i+1 i+2 i+3 i+4 i+5	Must use p_{1+1} or p_{i+2}	2 1 5 6 4 3 i-1 i i+1 i+2 i+3 i+4 i+5
$\text{new}(i+1,2)^3$	9 7 4 8 1 2 3 i i+1 i+2 i+3 i+4 i+5	At least 5	2 1 7 8 9 4 3 i-1 i i+1 i+2 i+3 i+4 i+5

Lemma 5.7. *For any on-line algorithm $\mathscr{A}$, a sequence $\sigma^2_{\mathscr{A}}$ exists such that*

$$l_{\mathscr{A}}(\sigma^1_{\mathscr{A}}\sigma^2_{\mathscr{A}}) \geqslant 5 \cdot \mathsf{opt}(\sigma^1_{\mathscr{A}}\sigma^2_{\mathscr{A}})/2$$

where $\sigma^1_{\mathscr{A}}$ is the sequence of Lemma 5.6.

Proof. From Lemma 5.6, we can restrict ourselves to the algorithms that with input $\sigma^1_{\mathscr{A}}$ pass through a configuration C_d, for some $d \geqslant 0$. We distinguish the following two cases:

(1) $d \leqslant 2$. Let us consider the case $d = 0$. Suppose three tasks 4, 5 and 6 of weight 2 are created in position $i+2$ (see the second row of Table 13). Then, one of these tasks (say 4) is assigned either to p_{i+1} or to p_{i+2} (otherwise the competitive ratio of $\mathscr{A}$ is 3). Suppose that tasks 5 and 6 terminate. When three new tasks of weight 2 arrive in position $i+1$ (see the third row) $\mathscr{A}$ yields a solution of measure 5. Notice that the optimum off-line solution is instead 2 (see the rightmost column of Table 13).

For C_1, we first consider the sequence $\mathtt{new}(i+1,2)^3$. Notice that, $\mathscr{A}$ must assign one of these tasks to p_{i+2} (otherwise the competitive ratio of $\mathscr{A}$ is 3). Let such a task be the surviving one, that is, we terminate the other two tasks of the above sequence. Then, we send sequence $\mathtt{new}(i+2,2)^3$ and, by a similar reasoning, another surviving task of weight 2 is assigned by $\mathscr{A}$ to p_{i+2}. Observe that, all the tasks can be assigned (off-line) in such a way that (1) p_{i-1} and p_{i+5} receive the unitary tasks of C_1, and (2) p_i and p_{i+1} receive the surviving tasks arising in $i+1$ and $i+2$, respectively. Hence, the optimum off-line is 2. Finally, we send the sequence $\mathtt{new}(i+3,2)^3$, which forces the algorithm to reach a load at least 5.

For C_2, we essentially generalize the proof for C_1. In particular, we first send two sequences: $\mathtt{new}(i+1,2)^3$ and $\mathtt{new}(i+4,2)^3$. Using the same argument, $\mathscr{A}$ must assign a surviving task to p_{i+2} and p_{i+3}. Then, the sequence $\mathtt{new}(i+2,2)^3$ will yield another surviving task either in p_{i+2} or in p_{i+3}. We complete the sequence

Table 14
Moving from C_d to C_{d-3}

Instance	On-line solution	Remark
new$(i+3, 1)^3$	1 2 I II III 3 4; i i+1 i+2 i+3 i+4 d-3 i+d+3	Otherwise we have C_3

with $\mathtt{new}(i+3,2)^3$: Since p_i+2 and p_{i+3} globally already received 3 tasks of weight 2 each, the best strategy in assigning the new tasks is to send two of them to p_{i+4}. This processors already had one task of weight 1, which yields a load equal to 5. Finally, it is easy to verify that the above sequence has optimum off-line equal to 2.

(2) $d>2$. In this case, we prove that $\mathscr{A}$ must pass through configurations C_{d-3} or configuration C_0. Let us consider configuration C_d with $d>2$ and let us suppose that three tasks arrive in position $i+3$ (see Table 14). If any of these tasks is assigned to processor p_{i+2}, then we easily obtain configuration C_0. Otherwise, both the two remaining processors p_{i+3} and p_{i+4} must be used. In this way we obtain the configuration C_{d-3}. By iterating this reasoning, we have that $\mathscr{A}$ passes through either C_0, C_1 or C_2.

The lemma thus follows. □

Theorem 5.8. *In the case of arbitrarily weighted tasks, any on-line algorithm has ratio at least* 2.5 *when* $k=1$.

Proof. It follows from Lemma 5.7. □

Let us observe that the 5/2 lower bound for the case $k=1$, given in Theorem 5.8, also hold for arbitrary k, since the same considerations apply in this case.

For $k>3$ the latter result can be improved so to show that the cluster algorithm is not too far from being optimal. The proof of this fact follows an approach similar to that used in the proof given in [4] for the case of temporary tasks on identical machines: indeed, we exploit a similar idea of creating several unitary tasks, followed by suitably weighted tasks.

Theorem 5.9. *For any* $k \geqslant 1$, *any on-line strategy is at least* $(3k+1)/(k+1)$-*competitive.*

Proof. Let us consider an interval of $4k+1$ processors and let us suppose that the instance starts by creating one task of weight $l=3k+1$ in each processor in the interval. It is then clear that either at least $2k+1$ of these tasks are assigned to the $2k+1$ central processors or at least $k+1$ tasks are assigned to one of the two external groups of k processors. Let us assume that the former case holds (the proof

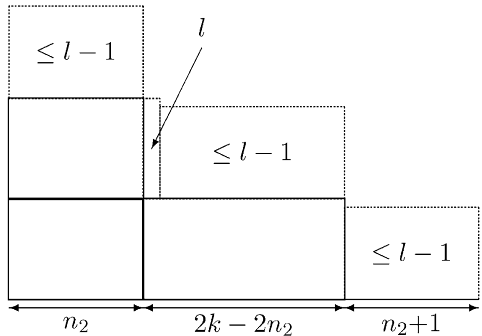


Fig. 4. The lower bound for $k \geqslant 1$.

in the latter case is very similar) and let us continue the instance by deleting the task that has been created in processor p_0 and all the tasks that have been assigned out of the central interval.

Without loss of generality, we assume that there exists $n_2 \geqslant 0$ such that the n_2 leftmost processors have been assigned 2 tasks, $2k - 2n_2$ central processors have been assigned 1 task, and the $n_2 + 1$ rightmost processors have been assigned no task. Observe that all these tasks can be assigned by an off-line solution to processors out of the central interval.

We now continue the instance by creating $2k(l+1)+1$ new unitary tasks in processor p_0. It is then easy to prove that one of the following three situations must occur: (a) one of the leftmost processors has a load at least equal to $3l$, (b) two central processors have a load at least equal to $2l$, (c) two right processors have a load at least equal to l, or (d) one central processor has a load at least equal to $2l$ and one right processor has a load at least equal to l. Indeed, let us assume (see Fig. 4) that no leftmost processor has a load at least equal to $3l$, at most one central processor has a load at least equal to $2l$, and no right processor has a load at least equal to l (the other case can be proved similarly). Then the total number of assigned tasks is at most

$$n_2(l-1) + l + (2k - 2n_2 - 1)(l-1) + (n_2+1)(l-1) = 2kl + l - 2k$$

while the total number of new tasks is

$$2kl + 2k + 1 > 2kl + l - 2k$$

since $l = 3k + 1 \leqslant 4k$. Thus, we have a contradiction.

Finally, it remains to show that any of the four situations (a)–(d) can be constructed so that the on-line solution is at least $(3k + 1)/(k + 1)$-competitive. Indeed, consider situation (d): in this case, we continue the instance by first deleting all the unitary tasks but those assigned to the two involved processors and by then creating $2k + 1$ tasks of weight l in processor p_0. By a simple counting argument, it follows that at least one processor must have a load at least equal to $3l$. On the other hand, the off-line solution could distribute both the unitary tasks and the tasks of weight l to all the $2k + 1$ central processors thus obtaining a load not greater than $l + 2$. Observe that, since

we are dealing with arbitrarily weighted tasks, the above argument implies that the competitive ratio of any on-line algorithm is at least $3l/(l+2)=(3k+1)/(k+1)$. □

6. Discussion and open problems

A connection with low earth orbit satellites. Further benefits of our approach and some consequences of our results deal with satellite low Earth orbit (LEO) constellations. In LEO constellations satellites are grouped into orbits so that they form a global coverage of the Earth's surface similar to that of cellular systems. However, due to the satellites movements, the footprints (and hence the cells) move with constant speed.

For that reason, handover occurs frequently because of the high speed of the satellites. An optimal strategy from this point of view is therefore that of assigning a user, which is located in the intersection of two or three cells, to the cell that guarantees the maximum time connection without handover (see, for example [26,27]). It is possible to see that, in the case of hexagonal grid model [11], *the optimal strategy for handover is exactly the cluster algorithm*. This observation combined with our results has the following unexpected consequence:

> Handover constraints forces us to perform a channel assignment strategy better than the greedy one.

Such a result is somehow counterintuitive, since one might think that spreading the load among all the available cells (regardless of handover constraints) allows to reduce the load in each cell. We finally remark that our approach *does not require additional communication among satellites* in order to decide to which satellite a new user has to be assigned.

Open problems. The first and more important open problem consists of closing the gap between the upper and the lower bound in the case of cellular networks. We conjecture that our algorithm is optimal. Notice that this would imply a *combinatorial* characterization of a class of instances that can be *on-line* solved without overcoming the cell capacity: in particular, this class includes all instances that admit an *off-line* assignment of maximum load equal to 1/4 of the capacity of a single cell. Each of these instances can be, in turn, characterized by using the results of [19]. Notice also that, for the one-dimensional case, we are already able to give such a characterization.

Another interesting research direction consists of studying the "permanent mobile tasks": in some cases, we have to deal with rapidly moving users so that, before one user ends its communication, it has changed position many times. In this case, we can ideally assume permanent tasks, i.e., once a task arrives it never terminates. This problem is actually something in between the on-line load balancing of permanent (non-mobile) tasks and our problem: It is a *special* case of temporary tasks with restricted assignment. In fact, the movement of a permanent task from one position to another can be simulated by terminating the task and creating a copy of it in the new position.

We think that our method also gives rise to a number of interesting questions concerning the solution of on-line problems, such as: (i) under which hypothesis is the cluster algorithm provably better than the greedy one? (ii) Can we give a (combinatorial) characterization of a set of on-line load balancing problems for which the cluster algorithm matches the lower bound? (iii) Can we use a decomposition into clusters to improve the competitive ratio of existing on-line algorithms other than the greedy one?

Finally, observe that the cluster algorithm applies to any bipartite graph corresponding to an instance of online load balancing. Since its performance depends on the cluster decomposition, an interesting open problem is that of efficiently constructing a good decomposition into clusters for a given bipartite graph.

References

[1] S. Albers, Better bounds for on-line scheduling, Proceedings of the 29th ACM Symposium on Theory of Computing (STOC), 1997, pp. 130–139.

[2] Y. Azar, On-line load balancing, in: A. Fiat, G. Woeginger (Eds.), On-line Algorithms—The State of the Art, Springer, Berlin, 1998.

[3] Y. Azar, A. Broder, A. Karlin, Online load balancing, Theoret. Comput. Sci. 130 (1994) 73–84.

[4] Y. Azar, L. Epstein, On-line load balancing of temporary tasks on identical machines, Proceedings of the Fifth Israeli Symposium on Theory of Computing and Systems (ISTCS), 1997, pp. 119–125.

[5] Y. Azar, B. Kalyanasundaram, S. Plotkin, K. Pruhs, O. Waarts, Online load balancing of temporary tasks, J. Algorithms 22 (1997) 93–110.

[6] Y. Azar, J. Naor, R. Rom, The competitiveness of online assignments, J. Algorithms 18 (1995) 221–237.

[7] A. Bar-Noy, A. Freund, J. Naor, On-line load balancing in a hierarchical server topology, SIAM J. Comput. 31 (2) (2001) 527–549.

[8] M.A. Bassiouni, C. Fang, Dynamic channel allocation for linear macrocellular topology, Proceedings of the ACM Symposium on Applied Computing (SAC), 1999, pp. 382–388.

[9] I. Caragiannis, C. Kaklamanis, E. Papaioannou, Efficient on-line communication in cellular networks, Proceedings of the 12th ACM Annual Symposium on Parallel Algorithms and Architectures (SPAA), 2000, pp. 46–53.

[10] S.K. Das, S.K. Sen, R. Jayaram, A dynamic load balancing strategy for channel assignment using selective borrowing in cellular mobile environment, ACM/Baltzer J. Wireless Networks 3 (5) (1997) 333–347.

[11] A. Ferreira, J. Galtier, P. Penna, Topological design, routing, and handover in satellite networks, in: I. Stojmenović (Ed.), Handbook of Wireless Networks and Mobile Computing, Wiley, New York, 2002, pp. 473–493.

[12] S. Fitzpatrick, J. Janssen, R. Nowakowski, Distributed online channel assignment for hexagonal cellular networks with constraints, Proceedings of the First International Workshop on Approximation and Randomization in Communication Networks (ARACNE), Proceedings in Informatics, Carleton Scientific Press, University of Waterloo, Waterloo, ON, Canada, 2000, pp. 147–154.

[13] P. Fizzano, D. Karger, C. Stein, J. Wein, Distributed job scheduling in rings, J. Parallel Distributed Comput. 45 (2) (1997) 122–133.

[14] R. Graham, Bounds for certain multiprocessor anomalies, Bell System Tech. J. 45 (1966) 1563–1581.

[15] R. Graham, Bounds on multiprocessor timing anomalies, SIAM J. Appl. Math. 17 (1969) 263–269.

[16] J. Janssen, D. Krizanc, L. Narayanan, S. Shende, Distributed online frequency assignment in cellular networks, J. Algorithms 36 (2000) 119–151.

[17] A.J. Kleywegt, V.S. Nori, M.W.P. Savelsbergh, C.A. Tovey, Online resource minimization, Proceedings of the 10th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), 1999, pp. 576–585.

[18] E. Kranakis, D. Krizanc, A. Pelc, Fault-tolerant broadcasting in radio networks, J. Algorithms 39 (2001) 47–67.

[19] D. Matula, M. Iridon, C. Yang, A graph theoretic approach for channel assignment in cellular networks, Wireless Networks 7 (6) (2001) 567–574.
[20] L. Narayanan, Y. Tang, Worst-case analysis of a dynamic channel assignment strategy, Proceedings of the ACM International Workshop on Discrete Algorithms and Methods for Mobile Computing (DIALM), 2000, pp. 8–17.
[21] K. Pahlavan, A. Levesque, Wireless Information Networks, Wiley-Interscience, New York, 1995.
[22] S. Phillips, J. Westbrook, Online load balancing and network flow, Algorithmica 21 (3) (1998) 245–261.
[23] A. Sen, T. Roxborough, S. Medidi, Upper and lower bounds of a class of channel assignment problems in cellular networks, Proceedings of IEEE INFOCOM'98, 1998.
[24] A. Sen, T. Roxborough, B.P. Sinha, On an optimal algorithm for channel assignment in cellular networks, Proceedings of the IEEE International Conference on Communications (ICC), 1999, pp. 1147–1151.
[25] E. Tardòs, J.K. Lenstra, D.B. Shmoys, Approximation algorithms for scheduling unrelated parallel machines, Math. Programming 46 (1990) 259–271.
[26] H. Uzunalioğlu, Probabilistic routing protocol for low earth orbit satellite networks, Proceedings of the IEEE International Conference on Communications (ICC), 1998.
[27] H. Uzunalioğlu, I.F. Akyildiz, Y. Yesha, W. Yen, Footprint handover rerouting protocol for low earth orbit satellite networks, Wireless Networks 5 (1999) 327–337.