Algorithmica
DOI 10.1007/s00453-008-9241-8

A Constant-Competitive Algorithm for Online OVSF Code Assignment

F.Y.L. Chin · H.F. Ting · Y. Zhang

Received: 15 December 2007 / Accepted: 8 October 2008

Abstract Orthogonal Variable Spreading Factor (OVSF) code assignment is a fundamental problem in Wideband Code-Division Multiple-Access (W-CDMA) systems, which plays an important role in third generation mobile communications. In the OVSF problem, codes must be assigned to incoming call requests with different data rate requirements, in such a way that they are mutually orthogonal with respect to an OVSF code tree. An OVSF code tree is a complete binary tree in which each node represents a code associated with the combined bandwidths of its two children. To be mutually orthogonal, each leaf-to-root path must contain at most one assigned code. In this paper, we focus on the online version of the OVSF code assignment problem and give a 10-competitive algorithm (where the cost is measured by the total number of assignments and reassignments used). Our algorithm improves the previous $O(h)$-competitive result, where h is the height of the code tree, and also another recent constant-competitive result, where the competitive ratio is only constant under amortized analysis and the constant is not determined. We also improve the lower bound of the problem from $3/2$ to $5/3$.

Keywords Wideband code-division multiple-access · Online algorithms · Competitive analysis · Lower bounds

The research of F.Y.L. Chin was supported by HK RGC grant HKU-7113/07E.
The research of H.F. Ting was supported by HK RGC grant HKU-7045/02E.

F.Y.L. Chin · H.F. Ting (✉) · Y. Zhang
Department of Computer Science, The University of Hong Kong, Pokfulam road, Hong Kong, Hong Kong
e-mail: hfting@cs.hku.hk

F.Y.L. Chin
e-mail: chin@cs.hku.hk

Y. Zhang
e-mail: yzhang@cs.hku.hk

1 Introduction

Wireless communication based on Frequency Division Multiplexing (FDM) technology is widely used in the area of mobile computing today and the frequency or channel assignment problem has been extensively studied for the cellular network [1–3, 5, 9]. Wideband Code-Division Multiple-Access (W-CDMA) technology is one of the main technologies widely-developed in recent years for the implementation of third-generation (3G) cellular systems. We consider the well-studied problem of Orthogonal Variable Spreading Factor (OVSF) code assignment in W-CDMA systems [6, 8, 10, 11, 13].

OVSF is an implementation of CDMA wherein, before each signal is transmitted, the spectrum is spread according to a unique code, which is derived from an OVSF code tree. An OVSF code tree is a complete binary tree. Users have requests for different data rates, and the OVSF code tree accommodates these different requests by assigning codes at different levels of the code tree, with the root being at the highest level and representing the entire bandwidth of the wireless system. The code at any node other than the root denotes the bandwidth half that of its parent in the tree. For any *legal assignment* in the code tree, no two assigned codes lie on a single path from the root to a leaf, i.e., any two assigned codes are *mutually orthogonal*. The subset of nodes in the code tree, which forms a legal assignment, is called a *code assignment*. A node x is said to be *free* if there are no assigned nodes in every root-to-leaf path containing x, and thus making x an assigned node would still result in a legal assignment. For convenience, we use the words "code" and "node" interchangeably. Figure 1 shows a legal assignment of an OVSF code tree in which a, b, c, d, e, f and g are assigned nodes. Note that h, i, j and k are the only free nodes in the assignment.

To illustrate the essence of the OVSF code assignment problem, consider the assignment in Fig. 1. Let $Req(x)$ denote the request to which code x is assigned. Suppose a level-0 code request arrives followed by a level-1 code request. If code i were assigned to the first request, we would have to make a code reassignment before we can assign code k to the second request (i.e., assign code h to $Req(i)$ and thereby freeing i). If, on the other hand, h were assigned to the first request, then we can assign code k to the second request without any reassignment.

Since each reassignment requires processing overhead and may affect the quality of communications, a natural idea is to design algorithms to minimize the number of reassignments. Note that this problem will not be too difficult and can be solved

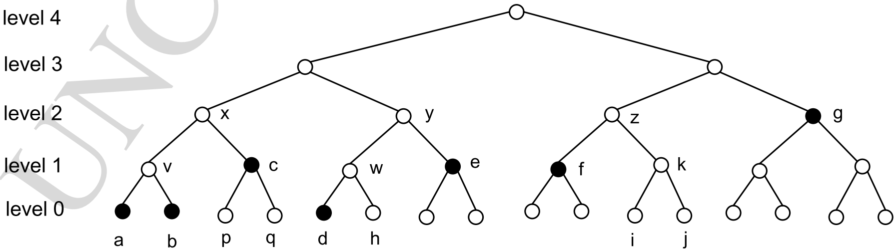


Fig. 1 Example of a legal assignment

optimally by a greedy strategy if codes were never released (see [6]). However, when codes can be released, the code tree can be fragmented and many reassignments might be needed if a good assignment algorithm was not used.

In general, an algorithm for OVSF code assignment is expected to handle a sequence $\sigma = (C_1, C_2, \ldots, C_k, \ldots)$ of code operations over time, each operation C_k being either to *request* a code at a particular level or to *release* an assigned code. Note that, if the total bandwidth of free codes is less than the bandwidth required by a code request, the new code request has to be withdrawn.

The OVSF code assignment problem is difficult, and the approach has often been to produce heuristics [8, 10, 13], whose performance is measured by the approximation (or competitive) ratio, which compares the cost of the algorithm to the cost of the optimal off-line scheme. Here, the cost is measured by the total number of assignments and reassignments made by the algorithm. The problem has been studied extensively recently. We list below three variations of this problem; they all assume that the code tree is of fixed height.

One-Step Off-line Code Assignment Given a code assignment A and a new level-i code request r, find a code assignment A', which satisfies the new request with a minimal number of reassignments. For this variation, Minn and Siu [10] gave a greedy algorithm, and Erlebach *et al.* [6] proved that this problem is NP-hard and gave an $O(h)$-approximation algorithm, where h is the height of the OVSF code tree.

General Off-line Code Assignment Given a sequence σ of code operations, find a sequence of code assignments and reassignments such that the total number of reassignments is minimum, assuming the initial code tree is empty. This variation was proved to be NP-hard by Tomamichel [12], who also gave an exponential-time algorithm for this variation.

Online Code Assignment The operations $C_1, C_2, C_3, \ldots$ in the sequence $\sigma = (C_1, C_2, \ldots, C_t, \ldots)$ arrive through time. At any time $t > 0$, we only know about the operations until t and have no information about any future operations $C_{t'}$ with $t' > t$. Again, the problem is to find a sequence of code assignments and reassignments such that the total number of assignments and reassignments is minimum. For this variation, Erlebach *et al.* [6] gave an $O(h)$-competitive algorithm, where h is the height of the code tree. They also derived a lower bound of $3/2$ on the competitive ratio. With resource augmentation, which means the online algorithm is allowed to use more bandwidth than the optimal scheme, a 4-competitive algorithm with a double-sized code tree was given in [6]. Using $1/8$ extra bandwidth (less resource augmentation), Chin *et al.* [4] gave a 5-competitive algorithm. Recently, Forišek *et al.* [7] gave an online algorithm whose competitive ratio is constant under amortized analysis. In their paper, there is no estimate about the size of the constant and the worst case can still be $O(h)$.

In this paper, we focus on the online OVSF code assignment problem and aim at improving the $O(h)$-competitive algorithm given in [6]. We note that the algorithm forces the assigned codes in the OVSF code tree into a single fixed format, and there are two worst-case format-respecting configurations which make the performance of the algorithm poor, one which is bad (i.e. requires a reassignment at each level of the OVSF code tree) for a code request but good (i.e. constant reassignments) for a code release and the other which is bad for a code release but good for a code request. Interestingly, these two code configurations differ by only one code assignment (but differ much in their structure), and so there exists a sequence of alternating code requests and releases, each of which requires h code reassignments, and hence $O(h)$ for the competitive ratio. By introducing the idea of *partially assigned nodes*, we are able to relax the format requirement and this enables us to obtain a 10-competitive algorithm, improving the previous $O(h)$-competitive result [6] and the amortized $O(1)$-competitive result [7].

The rest of this paper is organized as follows. Sections 2 and 3 give the preliminaries and the basic idea of our algorithm. Section 4 describes our 10-competitive algorithm. Section 5 gives the correctness proof of the algorithm. A new lower bound of the problem, improving the bound from 3/2 to 5/3, is presented in Sect. 6. We give our conclusions in Sect. 7.

2 Preliminaries

Let T be an OVSF code tree with a legal assignment A. In our discussion, we assume that T is ordered and nodes at a particular level are ordered from left to right. We say that node u is *dead* if either it is assigned or at least one of its descendents is assigned. We say that a level ℓ is *compact* if any node at level ℓ that is to the left of some dead node at ℓ is also dead. For example, in Fig. 1, nodes p and q are not dead and hence level-0 is not compact. Nodes v, w, x, y and z are all dead and level-1 and level-2 are compact. Note that by definition, level-3 and level-4 are also compact even though they do not have any assigned node. We say that the assignment A is compact if all levels of T are compact. The following lemma suggests that the assigned nodes in a compact assignment are sorted; if we scan the OVSF code tree from left to right, the levels of the assigned nodes are non-decreasing.

Lemma 1 *Suppose that the legal assignment A is compact. Let u be an assigned node at level i and v be an assigned node at level j where $i < j$. Then, the level-j ancestor a_u of u must be to the left of v.*

Proof Since A is legal, a_u cannot be v. If a_u is to the right of v, the dead node u has some nodes to its left, namely the level-i descendents of v, that are not dead (see Fig. 2). It follows that level-i, and hence A, is not compact; a contradiction. □

Intuitively, we should make the assignments compact in order to fully utilize the bandwidth. There is a simple strategy to ensure compactness: To serve a level-ℓ code request r, we "append" it to the right-end of the list of dead nodes at ℓ, or more precisely, we assign to r the node u that is immediately after the last dead node (i.e., the rightmost dead node) at ℓ. It is obvious that the resulting assignment is also compact. However, it may not be legal; although u does not have any assigned descendent (because it is not dead before the update), it may have an assigned ancestor. To solve the problem, we distinguish two kinds of levels. Consider any level ℓ. Let u be the

« ALGO 0000 layout: Small Extended v.1.2 reference style: mathphys file: algo9241.tex (ELE) aid: 9241 doctopic: OriginalPaper class: spr-small-v1.1 v.2008/10/02 Prn:13/10/2008; 13:10 p. 5»

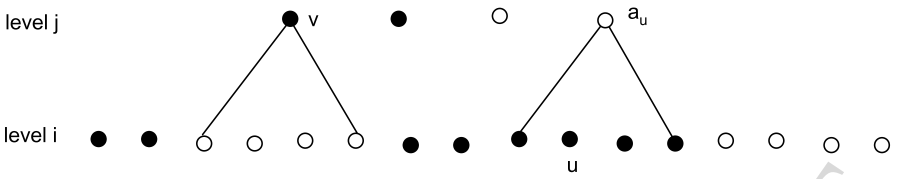
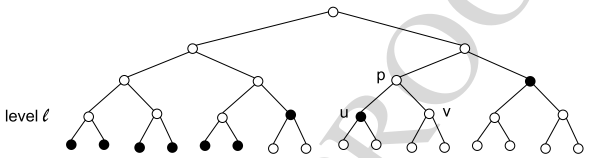


Fig. 2 Proof of Lemma 1

Fig. 3 Proof of Lemma 2



node immediately after the last dead node at ℓ (if ℓ does not have any dead node, let u be the leftmost node at ℓ).[1] We say that ℓ is *rich* if u is free, i.e., the node does not have any ancestor or descendent that is assigned; otherwise, ℓ is *poor*. For example, in Fig. 3, level ℓ is rich, and level $\ell - 1$ is poor. Note that a level may be poor even if no nodes at ℓ are assigned. It is easy to verify that if ℓ is rich, then the resulting assignment is still legal after assigning u to r. Suppose that ℓ is poor. Then, u is not free and it must have an ancestor v assigned to some request $Req(v)$. After assigning u to r, we need to reassign $Req(v)$, i.e., freeing v followed by a code request $Req(v)$, to make sure the assignment is legal. Note that this may trigger other reassignments of requests at higher levels.

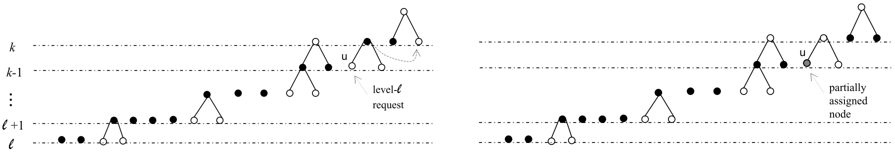
The following algorithm describes how this simple approach serves a level-ℓ request r. It makes use of two procedures AppendRich and AppendPoor, each of which makes exactly one node assignment. Let u be the node immediately after the last dead node at level ℓ (if there is no dead node at ℓ, then u is its leftmost node). Procedure AppendRich(ℓ, r) is used when ℓ is rich; it simply assigns u to request r. Procedure AppendPoor(ℓ, r) is for the case when ℓ is poor. After assigning u to r, AppendPoor(ℓ, r) frees the assigned ancestor a of u, and returns the request to which a is assigned before it is freed. In the description of the algorithm, we add a subscript to a request to indicate its level, e.g., the request r_g is a level-g request.

while ℓ is poor **do**
  $r_g = $ AppendPoor(ℓ, r); $\{r_g$ is a level-g request.$\}$
  $\ell = g$; $r = r_g$;
end while
AppendRich(ℓ, r);

[1] Note that we have ignored the case when all nodes at ℓ are dead because in such case, there is no more free bandwidth.

Journal ID: 453, Article ID: 9241, Date: 2008-10-13, Proof No: 1

« ALGO 0000 layout: Small Extended v.1.2 reference style: mathphys file: algo9241.tex (ELE) aid: 9241 doctopic: OriginalPaper class: spr-small-v1.1 v.2008/10/02 Prn:13/10/2008; 13:10 p. 6»



(a) Before the (re)assignments (b) After the (re)assignments

Fig. 4 The lazy approach

3 Some Ideas for Improvement

Note that the simple algorithm given in Sect. 2 might make a large number of calls to AppendPoor, and hence make a lot of (re)assignments, to serve a request. In the section, we give the ideas on how to reduce the number of (re)assignments. Then, we describe how to implement these ideas in Sect. 4.

The following lemma is the key for reducing the number of (re)assignments.

Lemma 2 *Suppose ℓ is poor. After executing* AppendPoor(ℓ, r), *ℓ becomes rich.*

Proof As shown in Fig. 3, if the last dead node u at ℓ is the left son of its parent p, then ℓ must be rich. It is because the right son v of p, which is immediately after u, must be free; v is not dead (because u is the last dead node) and hence has no assigned descendent, and u and v share the same set of ancestors and thus v does not have any assigned ancestor. It follows that if ℓ is poor, node u must be a right son. After AppendPoor(ℓ, r), the node after u, which is a left son, becomes the new last dead node of ℓ. As argued above, ℓ is now rich. □

Note that if we call AppendPoor m times, then m levels will be changed from poor to rich. This is good because the next time we serve any request on these rich levels, we just need a simple assignment. The problem is that there might be no more requests on these levels, and the effort is wasted. To solve the problem, we propose a lazy approach. Here is the idea. Suppose that there is a level-ℓ request r, and the levels $\ell, \ell+1, \ldots, k-1$ are all poor, and level k is rich. According to the simple approach, we will call AppendPoor $k-\ell$ times and then call AppendRich once. Our lazy approach will not make these $k-\ell$ calls for AppendPoor; instead, it jumps to the last step, calling AppendPoor to assign the node u after the last dead node at $k-1$ to the level-ℓ request r, followed by AppendRich for assigning a level-k node to the request assigned to the immediate ancestor (parent) of u (so as to free u).[2] (See Fig. 4).

Later, if there is a request on some level $g \in [\ell, k-1]$, we will do the necessary work that we have avoided previously in order to recover the correct free node at g and assign it to the request. To summarize, the lazy approach also aims at maintaining

[2]We are generous here by assigning a level-$(k-1)$ node to a level-ℓ request where $k > \ell$. If we insist that a level-ℓ node must be assigned to r, then we can actually assign a level-ℓ descendent of u to r.

Journal ID: 453, Article ID: 9241, Date: 2008-10-13, Proof No: 1

« ALGO 0000 layout: Small Extended v.1.2 reference style: mathphys file: algo9241.tex (ELE) aid: 9241 doctopic: OriginalPaper class: spr-small-v1.1 v.2008/10/02 Prn:13/10/2008; 13:10 p. 7»

compact assignments. However, there may be some nodes that are assigned to some lower-level requests; we call these nodes *partially assigned nodes*. These partially assigned nodes induce some structures called *tanks of free nodes*, or simply *tanks*, which are intervals $[\ell, k-1]$ of levels with the following properties: The levels $\ell, \ell+1, \ldots, k-2$ are all poor and the assigned nodes at these levels are all fully assigned (i.e., not partially assigned). For level $k-1$, its last dead node is a left son and partially assigned to a level-ℓ request, and the remaining assigned nodes are fully assigned.

To define tanks formally, we say that a level ℓ is *locally rich* if the last dead node at ℓ is a left son of its parent. The following fact is easy to verify from the definition and the proof of Lemma 2.

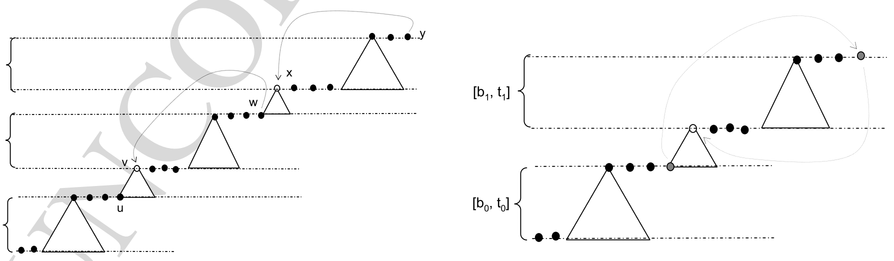
Fact 1 *A locally rich level is a rich level. Suppose ℓ is locally rich. Then after executing* AppendRich(ℓ, r), *ℓ is no longer locally rich. If ℓ is poor, then after executing* AppendPoor(ℓ, r), *ℓ becomes locally rich.*

Formally, a *tank* $[b, t]$ represents the levels between b and t where all levels from b to $t-1$ are poor, level t is locally rich and the last dead node at level t (which, by the definition of locally rich, must be a left son) is partially assigned to some request on level b. We say that b is the bottom of the tank $[b, t]$, and t is its top.

It is not difficult to implement the lazy approach in such a way that the number of (re)assignments needed for serving a request can be reduced substantially. However, to achieve a constant number of (re)assignments, we need to impose an extra structural property on tanks. Consider two tanks $[b_o, t_o]$ and $[b_1, t_1]$. Suppose that $[b_o, t_o]$ is below $[b_1, t_1]$, i.e, $t_o < b_1$. We say that the two tanks are *merge-able* if

(i) all the levels $t_o+1, t_o+2, \ldots, b_1-1$ between $[b_o, t_o]$ and $[b_1, t_1]$ are empty, i.e., the levels do not have any assigned nodes, and
(ii) the last dead node at t_o is the leftmost level-t_o descendent of its level-b_1 ancestor.

See Fig. 5b for an example. We find that merge-able tanks are bad for our approach. For example, in Fig. 5a, there are three tanks that are merge-able. If the request at node u is released, then node v is no longer dead. In order to preserve compactness, we need to move the request at the partially assigned node w to v. Then, x is no



(a) Problem with merge-able tanks

(b) Merging of two tanks

Fig. 5 Merge-able tanks

Journal ID: 453, Article ID: 9241, Date: 2008-10-13, Proof No: 1

longer dead, and we need to move the request at the partially assigned node y to x. In general, if there is a large number of consecutive tanks that are merge-able, we may need a large number of reassignments after a code release. To avoid such scenario, our updating procedure will merge any two merge-able tanks as soon as they appear. In other words, our algorithm keeps the following invariant:

$(*)$ There are no merge-able tanks in the assignment.

As can be seen in Fig. 5b, we can merge two merge-able tanks using two (re)assignments, and the merging preserves the compactness of the assignment.

4 A Lazy Algorithm

In this section, we describe the algorithm LAZY, which implements the lazy approach efficiently. To simplify the description, we regard a locally rich level ℓ that does not belong to any tank as a tank $[\ell, \ell]$ itself. Again, in our description, we will add a subscript to a request to indicate its level, e.g., the request r_g is a level-g request. In addition to AppendPoor and AppendRich, LAZY also makes use of the following two procedures.

- `FreeTail`(ℓ): The level ℓ must not be empty (i.e., must contain some assigned nodes.) The procedure frees the last assigned node u at ℓ and returns the request to which u is assigned.
- `ReAssign`(r, r'): Here, r is a request in the assignment, and the request r' is not in the assignment. Suppose that u is assigned to r. Then, the procedure frees u from r, and then assigns u to r'.

It is easy to see that `ReAssign` requires one node assignment while `FreeTail` does not require any. We are now ready to describe LAZY. We first describe how it serves a code request. Then, we explain how it releases a code. We assume that the assignment is legal and compact before the update.

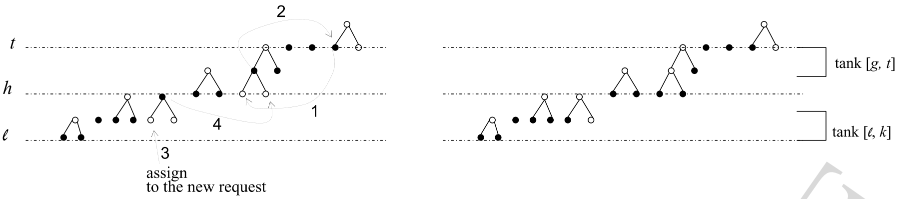
4.1 Serving a Level-ℓ Request r_ℓ

We have three different cases to consider.

Case 1 ℓ is poor and does not belong to any tank. If all levels above ℓ are poor, we report not enough bandwidth (and we will prove in Sect. 5 that this is true). Otherwise, it can be verified that there must be levels above ℓ that are either rich, or belong to some tanks. Let h be the lowest level above ℓ that either belongs to some tank, or is rich. Note that if h does not belong to any tank, then h is rich but not locally rich (otherwise, $[h, h]$ itself is a tank). In such case, we simply call AppendRich(h, r_ℓ). The case when h belongs to some tank is more complicated. In such case, h must be the bottom of some tank $[h, t]$. By definition, the last assigned node at level t is partially assigned to a level-h request r_h. Roughly speaking, to serve r_ℓ, we first recover the free node at h by re-assigning r_h back to a node at level h. Then, we insert r_ℓ to a level lower than h so that the (re)assignments can make use of the free node at h and stop. We give below the pseudo-code.

Journal ID: 453, Article ID: 9241, Date: 2008-10-13, Proof No: 1

« ALGO 0000 layout: Small Extended v.1.2 reference style: mathphys file: algo9241.tex (ELE) aid: 9241 doctopic: OriginalPaper class: spr-small-v1.1 v.2008/10/02 Prn:13/10/2008; 13:10 p. 9»



(a) Before the (re)assignments (b) After the (re)assignments

Fig. 6 The sequence of (re)assignments for Case 1

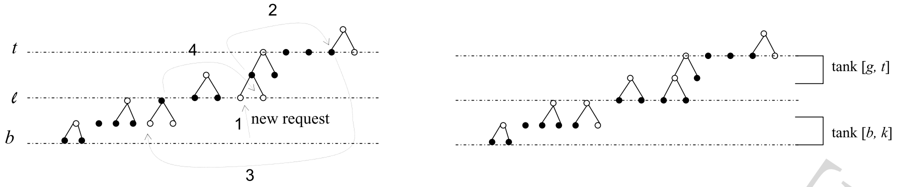
1: **if** all levels above ℓ are poor **then**
2: return and report "Not enough bandwidth";
3: **end if**
4: Let h be the lowest level above ℓ that either belongs to some tank or is rich.
5: **if** h does not belong to any tank **then**
6: AppendRich(h, r_ℓ); $\{[\ell, h]$ becomes a tank.$\}$
7: return;
8: **end if**
9: {The following code handles the case when h belongs to some tank and Fig. 6 shows graphically how the (re)assignments are done.}
10: Suppose that h belongs to the tank $[h, t]$.
11: **if** $h \neq t$ **then**
12: $r_h = \texttt{FreeTail}(t)$;
13: $r_g =$ AppendPoor(h, r_h); AppendRich(t, r_g); $\{[g, t]$ becomes tank.$\}$
14: **end if**
15: {At this point, h is not empty and is locally rich (Fact 1).}
16: Let k be the highest level below h that is not empty;
17: {If all levels below h are empty, let $k = 0$.}
18: **If** $(k < \ell)$ **then** let $k = \ell$;
19: $s =$ AppendPoor(k, r_ℓ); $\{s$ must be from h and $[\ell, k]$ becomes tank.$\}$
20: AppendRich(h, s);
21: {From Fact 1, h is not locally rich now and thus $[h, h]$ is not a tank.}

To ensure Invariant $(*)$, we need to do some tank mergings. For the case when h does not belong to any tank (lines 5–8), we need at most two tank-mergings, which make at most 4 reassignments. For the case when h belongs to some tank (lines 10–21), note that the additional tanks $[\ell, k]$ and $[g, t]$ may be created. As pointed out in line 21, $[h, h]$ is not a tank. Furthermore, there is no merge-able tank above $[g, t]$ (because $(*)$ ensures there is none above $[h, t]$ before the update). Therefore, we only need to merge tank below $[\ell, k]$, which requires two extra (re)assignments. Thus, we need at most 6 assignments in total to serve request r_ℓ. However, if we need to merge $[\ell, k]$ with a tank below, we can save the assignment used by AppendPoor(k, r_ℓ) at line 19; r_ℓ will be reassigned during the merging. This reduces the number of (re)assignments to 5.

Case 2 ℓ is poor and belongs to some tank $[b, t]$. For this case, we insert r_ℓ to ℓ. The tank $[b, t]$ may be broken into two tanks, one above, and one below ℓ.

Journal ID: 453, Article ID: 9241, Date: 2008-10-13, Proof No: 1

« ALGO 0000 layout: Small Extended v.1.2 reference style: mathphys file: algo9241.tex (ELE) aid: 9241 doctopic: OriginalPaper class: spr-small-v1.1 v.2008/10/02 Prn:13/10/2008; 13:10 p. 10»



(a) Before the (re)assignments (b) After the (re)assignments

Fig. 7 The sequence of (re)assignments for Case 2

$r_g = \text{AppendPoor}(\ell, r_\ell)$; {$\ell$ becomes locally rich.}
$r_b = \texttt{FreeTail}(t)$; $\text{AppendRich}(t, r_g)$; {$[g, t]$ becomes tank.}
{We have served r_ℓ successfully, but there is no node assigned to r_b.}
if $b = \ell$ **then**
  AppendRich(ℓ, r_b); {Now, ℓ is not locally rich.}
else
  Let k be the highest level below ℓ that is not empty.
  {If all levels below h are empty, let $k = 0$.}
  If ($k < b$) **then** let $k = b$;
  $s = \text{AppendPoor}(k, r_b)$; {$s$ must be from ℓ and $[b, k]$ becomes tank.}
  AppendRich(ℓ, s); {Now, ℓ is not locally rich.}
end if

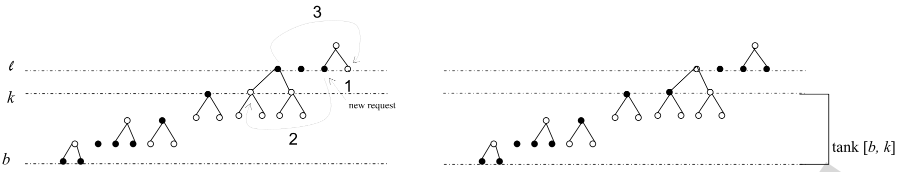
Figure 7 shows the sequence of (re)assignments for this case. Note that the total number of (re)assignments made is at most 4. Since ℓ is not locally rich, $[\ell, \ell]$ is not a tank. As ensured by ($*$), there is no merge-able tank above t or below b before the update, and thus the newly created tanks $[b, k]$ and $[g, t]$ do not need any merging.

Case 3 ℓ is rich. For this case, we do the following.

if ℓ does not belong to any tank **then**
  {The last dead node at ℓ must be a right son.}
  AppendRich(ℓ, r_ℓ);
  {ℓ becomes a tank and may need some tank mergings.}
else
  {Since ℓ is rich, it is the top of some tank $[b, \ell]$. See Fig. 8.}
  if $b = \ell$ **then**
    AppendRich(ℓ, r_ℓ);
  else
    $r_b = \texttt{FreeTail}(\ell)$; $\text{AppendRich}(\ell, r_\ell)$; {$\ell$ is still locally rich.}
    Let k be the highest level below ℓ that is not empty;
    If ($k < b$) **then** let $k = b$;
    $s = \text{AppendPoor}(k, r_b)$; {$s$ must be from ℓ and $[b, k]$ becomes tank.}
    AppendRich(ℓ, s); {Now, ℓ is not locally rich.}
  end if
end if

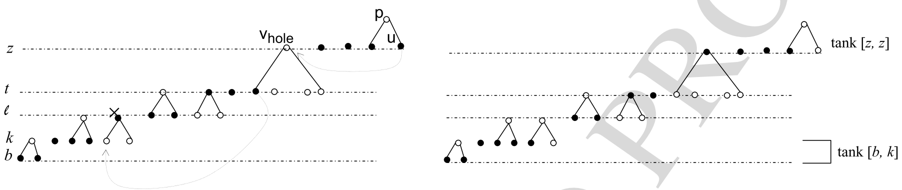
It can be verified that after the possible merging of tanks, the total number of (re)assignments made is at most 5.

Journal ID: 453, Article ID: 9241, Date: 2008-10-13, Proof No: 1



(a) Before the (re)assignments (b) After the (re)assignments

Fig. 8 The sequence of (re)assignments for Case 3



(a) Before the (re)assignments (b) After the (re)assignments

Fig. 9 The sequence of (re)assignments for release

4.2 Release of a Level-ℓ Node Assigned to Request r

We first consider the case when ℓ is not in any tank. Then, ℓ is not locally rich; otherwise, $[\ell, \ell]$ itself is a tank. For the operation of releasing r, we do the following:

1: $r_\ell = \texttt{FreeTail}(\ell)$; $\{[\ell, \ell]$ becomes a tank.$\}$
2: **if** $r \neq r_\ell$ **then** $\texttt{ReAssign}(r, r_\ell)$;

From the fact that ℓ is not locally rich before the update, it can be verified that the resulting assignment is still compact. Together with the two possible tank mergings (there may be tanks above or below $[\ell, \ell]$), the release needs at most 5 (re)assignments.

We now consider the case when ℓ is in some tank $[b, t]$. To release r, we do the following (see Fig. 9):

1: $r_b = \texttt{FreeTail}(t)$;
2: **if** $b = \ell$ **then**
3:    **if** $r \neq r_b$ **then** $\texttt{ReAssign}(r, r_b)$;
4: **else**
5:    Let k be the highest level below ℓ that is not empty;
6:    **If** $(k < b)$ **then** let $k = b$;
7:    $s = \text{AppendPoor}(k, r_b)$; $\{s$ must be from ℓ, and $[b, k]$ becomes tank.$\}$
8:    **if** $r \neq s$ **then** $\texttt{ReAssign}(r, s)$;
9: **end if**

« ALGO 0000 layout: Small Extended v.1.2 reference style: mathphys file: algo9241.tex (ELE) aid: 9241 doctopic: OriginalPaper class: spr-small-v1.1 v.2008/10/02 Prn:13/10/2008; 13:10 p. 12»

Note that after the execution, the assignment may not be compact; we have freed the last assigned node at level t and did not reassign the node to any request. This may create some holes (i.e., free nodes) between dead nodes at some levels above t. In such case, we need to fill up the holes as follows. Let z be the lowest level above t that is not empty, and $v_{\mathtt{hole}}$ be the free node, if any, created at level z after the above release operation (see Fig. 9). Note that by $(*)$, there is no merge-able tank above $[b, t]$; this implies z is not in any tank, and it is not locally rich. The following two steps will restore the compactness of the assignment:

1: $r_z = \mathtt{FreeTail}(z)$; {$z$ is now locally rich.}
2: Assign $v_{\mathtt{hole}}$ to r_z.

It is important to note that freeing the last assigned node u at level z will not create any problem; since z is not locally rich, u must be the right son of its parent p. After freeing u and assigning $v_{\mathtt{hole}}$ to r_z, p is still dead because its left son is dead. We can argue that we use at most 5 (re)assignments as follows. Since $[b, t]$ is a tank, there is no tank above or below it. Therefore, if there is no hole created, 2 (re)assignments suffice. If the hole $v_{\mathtt{hole}}$ is created, we can see in Fig. 9 that three (re)assignments are needed, and since $[z, z]$ becomes a tank, we may need two additional reassignments to merge it with the tank above level z.

The following theorem summarizes our discussion in this section.

Theorem 1 *Let A be a compact assignment satisfying* $(*)$. *LAZY serves any code request or code release for A using at most* 5 *(re)assignments, and the resulting assignment is still legal, compact and satisfies* $(*)$. *Furthermore, LAZY is* 10*-competitive.*

Proof We have already verified in our description of LAZY that the algorithm uses at most 5 (re)assignments for a code request or a code release. We can argue that LAZY is 10-competitive as follows. Suppose that there are m_1 code requests and m_2 code releases. Obviously, $m_2 \le m_1$. To serve these requests and releases, LAZY makes at most $5m_1 + 5m_2 \le 10m_1$ (re)assignments. Note that an optimal algorithm has to make at least m_1 assignments for the m_1 code requests. It follows that LAZY is 10-competitive.

To see that the assignments maintained by LAZY are legal, note that except for the case when we assign a request to a hole (which we have argued in Sect. 4.2 that the resulting assignment is still compact), LAZY only uses the procedures AppendRich and AppendPoor to assign a node to a request. It uses AppendRich for a rich level, in which the node after the last dead node must be free. Since AppendRich assigns this free node to the request, the updated assignment is still legal. LAZY uses AppendPoor for a poor level, in which the node after the last dead node has one assigned ancestor a. Since AppendPoor would eventually release the request r at a, the assignment is still legal after executing AppendPoor (and LAZY will keep using AppendRich and AppendPoor to find a new node for r). Furthermore, the resulting assignment is compact because LAZY always appends a request after the last dead node of a level, and whenever we free a node, we will immediately assign it (or its ancestor) to some other request. It also satisfies $(*)$ because we do all the necessary tank mergings after each operation. □

Journal ID: 453, Article ID: 9241, Date: 2008-10-13, Proof No: 1

5 LAZY Fully Utilizes the Bandwidth

In this section, we prove that LAZY fully utilizes the bandwidth. More precisely, we prove that if LAZY cannot find an assignment to satisfy all the requests, then no assignment can satisfy these requests. Our analysis needs the following notion of *leaf capturing*.

> A node that is fully assigned captures all of its leaf descendents. A node that is partially assigned to a level-ℓ request captures its 2^ℓ leftmost leaf descendents.

For any node u, define $\mathit{Free}(u)$ to be the set of leaf descendents of u that are not captured. Intuitively, for the root γ, $\mathit{Free}(\gamma)$ is the remaining bandwidth not used by the current assignment. For any set X of nodes, define $\mathit{Free}(X) = \bigcup_{u\in X} \mathit{Free}(u)$ and $F(X) = |\mathit{Free}(X)|$. It is obvious that for a fixed set L of code requests, different legal assignments for L have the same value of $F(\{\gamma\})$. Recall that the leaves are at level 0, and the root is at the highest level.

Lemma 3 *Consider any compact assignment A. Suppose that the levels $p, p+1, \ldots, q$ are all poor. Let w be a node at level p such that* (i) *w is not dead* (*i.e., w is not assigned, and none of its descendents are assigned*), *and* (ii) *its level-$(q+1)$ ancestor is dead. Then $F(\{w\}) = 0$.*

Proof For any $\ell \ge p$, let a_ℓ denote the level-ℓ ancestor of w. Let z be the level such that $a_p = w, a_{p+1}, \ldots, a_z$ are not dead, and a_{z+1} is dead. Obviously $z \le q$ because we assume that a_{q+1} is dead. It follows that level z is poor. Let a'_z be the sibling of a_z. We consider two cases.

- a'_z is to the right a_z: a'_z is not dead because a_z is not dead and A is compact.
- a'_z is to the left a_z: a'_z cannot be dead either. Otherwise, it is the last dead node of level z (the following node a_z is not dead), and together with the fact that a'_z is the left son of its parent, level z is locally rich; a contradiction.

Hence, both children a'_z and a_z of a_{z+1} are not dead. But a_{z+1} is dead. It follows that a_{z+1} is assigned, and thus $F(\{a_{z+1}\})$, and hence $F(\{w\})$, equals zero. □

Lemma 4 *Let A be an assignment maintained by LAZY. For any level ℓ, let D_ℓ be the set of dead nodes at level ℓ. Then, $F(D_\ell) < 2^\ell$.*

Proof The lemma is obviously true for level 0. Suppose that it is true for all levels below ℓ, and we consider the level ℓ. Let $u_1, u_2, \ldots, u_k$ be the sequence of dead nodes at ℓ where u_k is the last dead node. First, we assume that there is no partially assigned node at ℓ. Let u_i be the last node that is not assigned. Then, $F(D_\ell) = F(\{u_1, u_2, \ldots u_i\}) + F(\{u_{i+1}, \ldots, u_k\})$. Since u_i is the last node not assigned, $u_{i+1}, \ldots, u_k$ are all assigned and the second term $F(\{u_{i+1}, \ldots, u_k\})$ is zero. To estimate the first term, we consider the last dead node w at level $\ell - 1$. Note that w cannot be a child of any node to the right of u_i because all nodes to the right of u_i are either assigned or not dead. On the other hand, it cannot be a child of $u_1, \ldots, u_{i-1}$; otherwise the two sons of u_i are not dead because w is the last dead node at $\ell - 1$,

Journal ID: 453, Article ID: 9241, Date: 2008-10-13, Proof No: 1

and together with the fact that u_i is not assigned, we conclude that u_i is not dead; a contradiction. Therefore, w must be a child of u_i. If w is the right child of u_i, then $F(\{u_1, \ldots, u_i\}) = F(D_{\ell-1})$; otherwise $F(\{u_1, \ldots, u_i\}) = F(D_{\ell-1}) + 2^{\ell-1}$. Together with the induction hypothesis that $F(D_{\ell-1}) < 2^{\ell-1}$, the lemma follows.

We now consider that case when there is a partially assigned node at ℓ. According to LAZY, the last dead node u_k is the only partially assigned node at ℓ. Suppose that it is partially assigned to a level-g request. Then, $[g, \ell]$ is a tank and the levels $g, g+1, \ldots, \ell-1$ are all poor. Let $w_1, w_2, \ldots, w_m$ be the sequence of level-g descendents of $u_1, u_2, \ldots, u_{k-1}$. Then, $F(D_\ell) = F(u_1, \ldots, u_{k-1}) + F(u_k) = F(w_1, \ldots, w_m) + F(u_k) = F(w_1, \ldots, w_i) + F(w_{i+1}, \ldots, w_m) + F(u_k)$ where w_i is the last dead node at level g. By the induction hypothesis, we conclude that $F(D_g) = F(\{w_1, \ldots, w_i\}) < 2^g$, and by the definition of captured leaves for partially assigned node, we have $F(u_k) = 2^\ell - 2^g$. Note that for any $w \in \{w_{i+1}, \ldots, w_m\}$, w is not dead, and its level-ℓ ancestor is dead, and by Lemma 3, $F(\{w_{i+1}, \ldots, w_m\}) = F(\{w_{i+1}\}) + \cdots + F(\{w_m\}) = 0$. The lemma follows. □

Theorem 2 *Suppose that LAZY reports "not enough bandwidth" when serving a level-ℓ request r. Let L be the set of requests in the current assignment. Then, there is no assignment that can satisfy all the requests in $L \cup \{r\}$.*

Proof LAZY reports "not enough bandwidth" because level ℓ, as well as all levels above ℓ are poor. Let $u_1, u_2, \ldots, u_i$ be the sequence of dead nodes, and $u_{i+1}, \ldots, u_m$ be the remaining nodes, at ℓ. Then there are $F(\{u_1, \ldots, u_i\}) + F(\{u_{i+1}, \ldots, u_m\})$ leaves that are not captured. By Lemma 4, we conclude that $F(\{u_1, \ldots, u_i\}) < 2^\ell$. For $F(\{u_{i+1}, \ldots, u_m\})$, let h be the level of the root. Since (i) levels $\ell, \ell+1, \ldots, h-1$ are poor, (ii) the single node at level h is dead, and (iii) the nodes $u_{i+1}, \ldots, u_m$ are not dead, we can apply Lemma 3 and conclude that $F(\{u_{i+1}, \ldots, u_m\}) = 0$. It follows that $F(\{u_1, u_2 \ldots, u_m\} < 2^\ell$. Since assigning any node to r needs to capture 2^ℓ leaves, there is no assignment that can satisfy all requests $L \cup \{r\}$. □

6 Lower Bound

In [6], it is shown that the competitive ratio of any online algorithm for the code assignment problem must be at least 1.5. The following theorem shows that this bound can be improved to 5/3 by modifying the main idea given in [6].

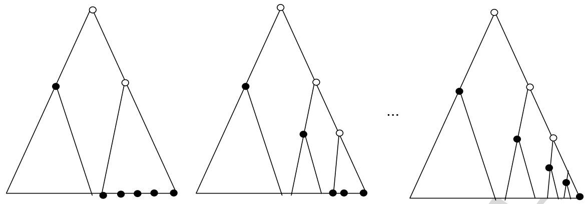
Theorem 3 *No deterministic algorithm can solve the online code assignment problem better than* 5/3*-competitive.*

Proof Consider a code tree with N leaves (leaf-codes) and a sequence of level-0 code requests with each request assigned to each leaf code one by one. As soon as both right and left subtrees of the root have at least $N/4$ assigned leaf codes, the adversary will stop issuing any more level-0 code requests. Thus there will be $R \leq 3N/4$ level-0 code requests in the sequence. Then the adversary will repeatedly release those requests in the subtree with more than $N/4$ assigned leaf codes until

Journal ID: 453, Article ID: 9241, Date: 2008-10-13, Proof No: 1

« ALGO 0000 layout: Small Extended v.1.2 reference style: mathphys file: algo9241.tex (ELE) aid: 9241 doctopic: OriginalPaper class: spr-small-v1.1 v.2008/10/02 Prn:13/10/2008; 13:10 p. 15»

Fig. 10 The recursive construction for deriving the lower bound



both subtrees have exactly $N/4$ assigned leaf codes. The adversary will then make a level-($(\log N) - 1$) request which will cause at least $N/4$ code reassignments, which end up with either the right or the left subtree with full assigned leaf codes. The adversary will then proceed recursively with the subtree with full assigned leaf codes by releasing its every other node. This process will be repeated $\log_2 N - 1$ times with a total of $N/2 - 1$ reassignments (See Fig. 10).

On the other hand, the optimal algorithm can assign the leaf codes in such a way that no extra reassignments will be needed. Thus the optimal algorithm will make $R + \log_2 N - 1$ assignments, whereas the online algorithm will take a total of $R + N/4 + N/8 + \cdots + 1 + \log_2 N - 1 = R + N/2 + \log_2 N - 2$ (re)assignments. Since $R \leq 3N/4$, the competitive ratio

$$\frac{R + N/2 + \log_2 N - 2}{R + \log_2 N - 1}$$

will be no less than $5/3$ (asymptotically). □

7 Conclusions

We have given in this paper the first constant competitive (worst case) algorithm for the online OVSF code assignment problem; it uses at most 5 (re)assignments to serve a code request or to code release. As mentioned in Sect. 1, our algorithm is based on the observation that the fixed format enforced by the $O(h)$-competitive algorithm of Erlebach *et al.* [6] is too stringent; it results two worst-case format-respecting configurations in its assignments, one is good for a code request and bad for a code release and the other is good for a code release and bad for a code request. By introducing the idea of partially assigned nodes, we are able to relax this format requirement such that the assignments maintained by our algorithm has only one worst-case configuration. This makes our algorithm much more competitive. However, we note that our format still requires the assignments to satisfy some global structural properties. To make further improvement, we believe that a promising direction is to find instead some local structural properties of an assignment that can still guarantee full utilization of the bandwidth. It is likely that fewer (re)assignments are needed to preserve local properties and hence we may have a more competitive algorithm.

Acknowledgements The authors thank M.Y. Chan for her efforts in making this paper more readable, and the reviewers for their very helpful comments.

Journal ID: 453, Article ID: 9241, Date: 2008-10-13, Proof No: 1

References

1. Chan, W.T., Chin, F.Y.L., Ye, D., Zhang, Y., Zhu, H.: Frequency allocation problem for linear cellular networks. In: Proceedings of the 17th Annual International Symposium on Algorithms and Computation. Lecture Notes in Computer Science, vol. 4288, pp. 61–70. Springer, Berlin (2006)
2. Chan, W.T., Chin, F.Y.L., Ye, D., Zhang, Y., Zhu, H.: Greedy online frequency allocation in cellular networks. Inf. Process. Lett. **102**(2–3), 55–61 (2007)
3. Chan, W.T., Chin, F.Y.L., Ye, D., Zhang, Y., Zhu, H.: Online frequency allocation in cellular networks. In: Proceedings of the 19th ACM Symposium on Parallelism in Algorithms and Architectures, pp. 241–249 (2007)
4. Chin, F.Y.L., Zhang, Y., Zhu, H.: Online OVSF code assignment in cellular networks. In: Proceedings of the 3rd International Conference on Algorithmic Aspects in Information and Management. Lecture Notes in Computer Science, vol. 4508, pp. 191–200. Springer, Berlin (2007)
5. Caragiannis, I., Kaklamanis, C., Papaioannou, E.: Efficient on-line frequency allocation and call control in cellular networks. Theory Comput. Syst. **35**(5), 521–543 (2002). A preliminary version of the paper appeared in SPAA 2000
6. Erlebach, T., Jacob, R., Mihalak, M., Nunkesser, M., Szabo, G., Widmayer, P.: An algorithmic view on OVSF code assignment. Algorithmica **47**, 269–298 (2007)
7. Forišek, M., Katreniak, B., Katreniaková, J., Královič, R., Královič, R., Koutný, V., Pardubská, D., Plachetka, T., Rovan, B.: Online bandwidth allocation. In: Proceedings of the 16th Annual European Symposium on Algorithms. Lecture Notes in Computer Science, vol. 4698, pp. 546–557. Springer, Berlin (2007)
8. Li, X.Y., Wan, P.J.: Theoretically good distributed CDMA/OVSF code assignment for wireless ad hoc networks. In: Proceedings of the 11th Annual International Conference of Computing and Combinatorics, pp. 126–135 (2005)
9. McDiarmid, C., Reed, B.A.: Channel assignment and weighted coloring. Networks **36**(2), 114–117 (2000)
10. Minn, T., Siu, K.Y.: Dynamic assignment of orthogonal variable-spreading factor codes in W-CDMA. IEEE J. Sel. Areas Commun. **18**(8), 1429–1440 (2000)
11. Rouskas, A.N., Skoutas, D.N.: OVSF codes assignment and reassignment at the forward link W-CDMA 3G systems. In: Proceedings of the 13th IEEE International Symposium on Personal, Indoor and Mobile Radio Communications
12. Tomamichel, M.: Algorithmische Aspekte von OVSF Code Assignment mit Schwerpunkt auf Offline Code Assignment. Student thesis at ETH Zürich (2004)
13. Wan, P.J., Li, X.Y., Frieder, O.: OVSF-CDMA code assignment in wireless ad hoc networks. In: Proceedings of DIAL M-POMC 2004 Joint Workshop on Foundations of Mobile Computing, pp. 92–101 (2004)