Online Bandwidth Allocation*

Michal Forišek[1], Branislav Katreniak[1], Jana Katreniaková[1], Rastislav Královič[1], Richard Královič[1,2], Vladimír Koutný[1], Dana Pardubská[1], Tomáš Plachetka[1], and Branislav Rovan[1]

[1] Dept. of Computer Science, Comenius University, Mlynská dolina, 84248 Bratislava, Slovakia.
[2] Dept. of Computer Science, ETH Zürich, Switzerland

Abstract. The paper investigates a version of the resource allocation problem arising in the wireless networking, namely in the OVSF code reallocation process. In this setting a complete binary tree of a given height n is considered, together with a sequence of requests which have to be served in an online manner. The requests are of two types: an insertion request requires to allocate a complete subtree of a given height, and a deletion request frees a given allocated subtree. In order to serve an insertion request it might be necessary to move some already allocated subtrees to other locations in order to free a large enough subtree. We are interested in the worst case average number of such reallocations needed to serve a request.
In [4] the authors delivered bounds on the competitive ratio of online algorithm solving this problem, and showed that the ratio is between 1.5 and $O(n)$. We partially answer their question about the exact value by giving an $O(1)$-competitive online algorithm.
In [3], authors use the same model in the context of memory management systems, and analyze the number of reallocations needed to serve a request in the worst case. In this setting, our result is a corresponding amortized analysis.

1 Introduction and motivation

Universal Mobile Telecommunications System (UMTS) is one of the third-generation (3G) mobile phone technologies that uses W-CDMA as the underlying standard, and is standardized by the 3GPP [8]. The W-CDMA (Wideband Code Division Multiple Access) is a wideband spread-spectrum 3G mobile telecommunication air interface that utilizes code division multiple access. The main idea behind the W-CDMA is to use physical properties of interference: if two transmitted signals at a point are in phase, they will "add up" to give twice the amplitude of each signal, but if they are out of phase, they will "subtract" and give a signal that is the difference of the amplitudes. Hence, the signal received by a particular station is the sum (component-wise) of the respective transmitted vectors of all senders in the area. In the W-CDMA, every sender s is given

* The research has been supported by grant APVV-0433-06 and VEGA 1/3106/06.

a *chip code* $\mathbf{v}$. Let us represent the data to be sent by a vector of ± 1. When s wants to send a *data vector* $\mathbf{d} = (d_1, \ldots, d_n)$, $d_i \in \{1, -1\}$, it sends instead a sequence $d_1 \cdot \mathbf{v}, d_2 \cdot \mathbf{v}, \ldots, d_n \cdot \mathbf{v}$, i.e. n-times the chip code modified by the data. For example, consider a sender with a chip code $(1, -1)$ that wants to send data $(1, -1, 1)$; then the actually transmitted signal is $(1, -1, -1, 1, 1, -1)$. The signal received by a station is then a sum of all transmitted signals. Clearly, if the chip codes are orthogonal, it is possible to uniquely decode all the signals.

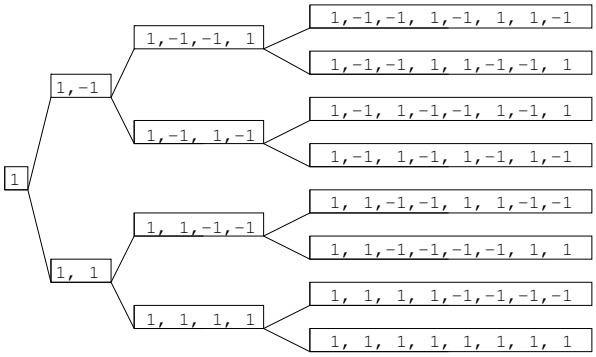

Fig. 1. An OVSF tree

One commonly used method of implementing the chip code allocation is Orthogonal Variable Spreading Factor Codes (OVSF). Consider a complete binary tree, where the root is labeled by (1) , the left son of a vertex with label α is labeled (α, α) and the right son is labeled $(\alpha, -\alpha)$ (see Figure 1).

If a sender station enters the system, it is given a chip code from the tree in such a way that there is at most one assigned code from each root-to-leaf path. It can be shown [1, 7] that this construction fulfills the orthogonality property even with codes of different lengths.

Clearly, a code at level l in the tree has length 2^l, and a sender using this code will use a fraction of $1/2^l$ of the overall bandwidth. When users enter the system, they request a code of a given length. It is irrelevant which particular code is assigned to which user, the length is the only thing that matters. When users connect to and disconnect from a given base station, i.e. request and release codes, the tree can become fragmented. It may happen that no code at the requested level is available, even though there is enough bandwidth (see Figure 2).

This problem can be solved by changing the chip codes of some already registered users, i.e. reallocating the vertices of the tree. Since the cost of a reallocation dominates this operation, the number of reallocations should be kept minimal. In [4] the authors considered the problem of minimizing the number of reallocations over a given sequence of requests and showed that it is NP-hard to generate an optimal allocation schedule. In this paper, we show that the online

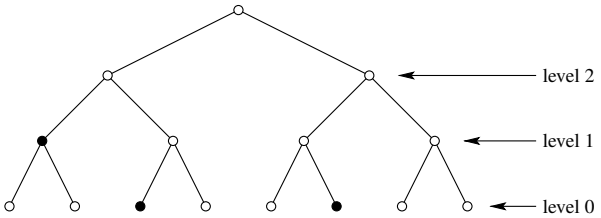

Fig. 2. No code at level 2 can be allocated although there is enough free bandwidth. Full circles represent allocated vectors.

version proposed in [4] can be solved in amortized complexity $O(1)$ reallocations per request. Due to space restrictions some technical parts have been omitted from this paper, and can be found in the technical report [5].

2 Problem definition

Consider a complete binary tree $T = (V, E)$ of height n with leaves at level 0 and the root at level n. A *request vector* is a vector $\mathbf{r} = (r_0, \ldots, r_n) \in \mathbb{N}^{n+1}$, where r_i represents the number of users that request a code at level i. A *code assignment* of a particular request vector $\mathbf{r}$ is a subset of vertices $F \subset V$, such that every path from a leaf to the root contains at most one vertex from F, and there are exactly r_i vertices at level i in F.

The input consists of a sequence of requests of two types: *insertion* and *deletion*. Let $\mathbf{r}_t$ be the request vector and F_t its corresponding code assignment after t requests. The algorithm has to process the next request in the following way:

A) insertion request at a vertex at level i.
The algorithm must output a new code assignment F_{t+1} satisfying the request vector $\mathbf{r}_{t+1} = (r_0, \ldots, r_i + 1, \ldots, r_n)$[3]

B') deletion request of a particular vertex $v \in F_t$.
Let v be at level i. The new request vector is $\mathbf{r}_{t+1} = (r_0, \ldots, r_i - 1, \ldots, r_n)$, and the new code assignment is $F_{t+1} = F_t \setminus \{v\}$.

During each step, the number of reassignments is $|F_{t+1} \setminus F_t|$. For a given sequence of requests $R_1, \ldots, R_m$, we are interested in the amortized number of reassignments per request, i.e. the quantity $\frac{1}{m} \sum_{t=0}^{m-1} |F_{t+1} \setminus F_t|$.

Note that there is no harm in allowing the reallocations within the deletion requests. In a deletion request the algorithm remembers the moves it would perform, and performs them in the next insertion request; in case of consecutive

[3] We may assume, without loss of generality, that there is enough bandwidth to satisfy each request.

deletion requests it is enough to maintain a mapping between the actual vertices in F and their "virtual" positions. This leads to a reformulation of a deletion request – instead of deleting the particular vertex v at level i, algorithm is allowed to delete *some* vertex at level i instead:

B) deletion request of a vertex at level i.
The algorithm is required to produce a new code assignment F_{t+1} satisfying the request vector $\mathbf{r}_{t+1} = (r_0, \ldots, r_i - 1, \ldots, r_n)$.

This definition bears resemblance to memory allocation problems studied in the operating systems community, in particular to the *binary buddy system* memory allocation strategy introduced in [6]. In this strategy, requests to allocate and deallocate memory blocks of sizes 2^l are served. The system maintains a list of free blocks of sizes of 2^k. When allocating a block of size 2^l in a situation where no block of this size is free, some bigger block is recursively split into two halves called *buddies*. When a block whose buddy is free is deallocated, both buddies are recursively recombined into a bigger block.

The properties of binary buddy system have been extensively studied in the literature (see e.g. [2] and references therein). However, the bulk of this research is focused on cases without reallocation of memory blocks. E.g. [2] shows how to implement the buddy system in amortized constant time per allocation/deallocation request, but in a model where reallocation is not allowed. As the restriction of not allowing reallocations makes the model substantially different, the results in [2] can not be directly applied to the model with reallocations allowed.

A binary buddy system with memory block reallocations has been studied in [3]. This paper analyzes both the number of reallocated blocks and the number of reallocated bytes per request; the analysis of the number of reallocated blocks is in fact the very same model that is used in our paper. However, only the worst case scenario of simple greedy algorithms is analyzed in [3] and presented results are quite pessimistic: $s - 1$ blocks have to be reallocated in worst case when allocating a block of size s. There are stronger results presented in [3], but all of them require a significant amount of auxiliary memory.

3 The algorithm

In this section we propose an online algorithm that processes the sequence of insertion/deletion requests according to the rules A) and B). The goal is to keep the amortized number of reallocations per request constant. We start with some notions and notations.

If we order the leaves from left to right and label them with numbers $0, \ldots, 2^n - 1$, there is an interval of the form $I_u = \langle i2^l, (i+1)2^l - 1 \rangle$ assigned to each tree vertex u of level l. We call such an interval a *place* of level l, and say that I_u begins at position $i2^l$. For a given code assignment F, a place I_u corresponding to a vertex u is called an *empty* (or *free*) *place* if neither u nor any vertex from the subtree rooted at u is in F. For $u \in F$ the corresponding place I_u is an *occupied*

place, and we say that there is a *pebble* of level l located on I_u (or, alternatively there is a pebble of level l at position $i2^l$). Since in a code assignment F, every path from a leaf to the root contains at most one vertex, we can view the code assignment F as a sequence of disjoint places which are either empty or occupied by pebble (see Figure 3). While the free places in this decomposition are not uniquely defined, we shall overlook this ambiguity, as we will argue either about a particular place or about the overall size of free places (called also *free bandwidth*).

We say that a pebble (or place) of level l has *size* 2^l. The left (right) neighbor of a pebble is a pebble placed on the next occupied place to the left (right). Analogously we talk about a left (right) neighbor of a free place. We denote pebbles by capital letters $A, B, \ldots, X$, and their corresponding sizes by $a, b, \ldots, x$. The notion of a vertex and the corresponding place are used interchangeably.

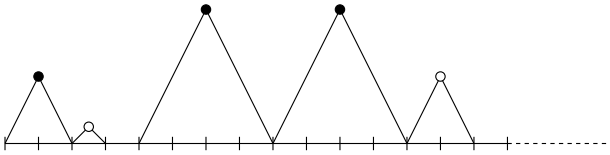

Fig. 3. A code assignment seen as a sequence of black and white pebbles.

The key idea of our algorithm is to maintain a well defined structure in the sequence of pebbles. An obvious approach would be to keep the sequence sorted – i.e. the pebbles of lower levels always preceding the pebbles of higher levels with only the smallest necessary free places between them. It is easy to see that the addition to and deletion from such a structured sequence can be done with at most $O(n)$ reallocations.[4] Unfortunately, it is also not difficult to see that there is a sequence of requests such that this approach needs amortized $O(n)$ reallocations per request (see [4]). The problem is that a too strictly defined structure needs too much "housekeeping" operations.

To overcome this problem we will keep the maintained structure less rigid. The pebbles are colored and kept so that the *black* pebbles form a sorted sequence according to the previous rule. There are several restrictions imposed on *white* pebbles. Informally, if there is a free place in the black sequence, a single white pebble can be placed at the beginning of this place. Moreover, all white pebbles form an increasing sequence. When moved, the pebble can change its color.

Formally, the structure of the sequence of pebbles is described by the following invariants.

[4] The addition request is processed by placing the pebble of level l at the end of sequence of pebbles of this level. If this place is already occupied by some pebble B, B is removed and reinserted. Since there are at most n different levels, and the levels of reinserted pebbles are increasing, the whole process ends after $O(n)$ iterations. The deletion works in a similar fashion.

C: Pebble A is *black*, if there is no bigger pebble before A, otherwise it is *white*.
P1: The free bandwidth before any pebble X is strictly less than x.
P2: There is always at least one black pebble between any two white pebbles.
P3: There is no white pebble between two black pebbles of the same size.

A sequence of pebbles and free places satisfying **C, P1**–**P3** will be called a *valid situation*. The key observation about valid situations is that in a valid situation it is always possible to process an insertion request without reallocations:

Lemma 1. *Consider a valid situation, and an insertion request of size $a = 2^l$. If there is a free bandwidth of at least a, then there is also a free place of size a.*

The algorithm is designed to maintain the valid situations over the whole computation, so all requests can be processed without reallocations. However, while processing a request, the intermediate situation might temporarily become not valid. The key to an effective algorithm is to develop a post-processing phase in each request that restores the validity using only a few reallocations. We show that in the case of insertion requests, a constant number of reallocations in each request is sufficient. In the deletion requests, however, a more involved accounting argument is used to show that the *average* number of reallocations per request in a worst-case execution remains constant.

Last notion connected to a structure of pebbles is that of *closing position*.

Definition 1. *The* closing position *of level l (of size x) is the position after the last black pebble of level l (of size x) if such a pebble exists. Otherwise, the closing position of level l is the position of the first pebble of higher level (size bigger than x).*

3.1 Procedure INSERT

We assume that whenever the procedure INSERT is called, it is to process a new request of size a in a valid situation in which there is a free bandwidth of size at least a. First, a free place of size a is found, and a pebble is put on this place. If the sequence is no longer valid after this operation, the algorithm reassigns a constant number of pebbles in order to restore the invariants. The procedure is listed as Algorithm 1, and its analysis is given in the following theorem:

Theorem 1. *Consider a valid situation, and an insertion request of size a. If there is a free bandwidth of size at least a, procedure* INSERT *correctly processes the request. Moreover, the output situation is valid, and only a constant number of pebbles has been reassigned within the execution.*

Sketch of proof. Here, we present an overall structure of the proof with a number of unproven claims. The complete version can be found in [5].

Consider an insertion request of size a, and suppose the invariants **C, P1**–**P3** hold. Let P be the first free place of size a; Lemma 1 ensures existence of such place. If P does not have a left neighbor (i.e. there is no other pebble present)

Algorithm 1 Procedure INSERT: inserts a pebble A of size a into a valid situation in which the free bandwidth of size $\geq a$ is guaranteed.

```
 1: let P be the first free place of size a
 2: let B be the left neighbor of P
 3: if B does not exist or B is black
    then
 4:     put A on P
 5:     return
 6: end if
 7:
 8: let C be the left neighbor of B
 9: if c < a then
10:     put A on P
11:     return
12: else if c = a then
13:     remove B
14:     put A just after C
15:     put B just after A
16:     return
17: end if
18:
19: remove B
20: if a < b then
21:     rename A and B so that A
        is the bigger pebble
22: end if
23:
24: let D be the pebble at the closing
    position of size a.
25: if D is white then
26:     let E be D's right neighbor
27:     remove D, E
28:     put A at D's original position
29:     put B after A
30:     put D after B
31:     put E at B's original position
32: else if D is black then
33:     remove D
34:     put A at D's original position
35:     put B after A, put D after C
36: end if
```

or its left neighbor B is black, then the algorithm puts the pebble A on P and the situation remains valid.

From now on suppose that P has a white left neighbor B, and denote the potential right neighbor of P by Z. However, there exists a left neighbor C of B, such that C is black and $c > b$. We distinguish three sub-cases: $c < a$, $c = a$, and $c > a$. When $c < a$, the possible right neighbor Z is black and its size is bigger than a. Therefore the black pebble A might be put on P resulting in a valid situation (line 10). If $c = a$, the sequence of lines 13–16 is executed: first, white pebble B is temporarily removed. Since there has been no other pebble between A and C, and $a = c$, the place of size a immediately following C is now free and black A is placed immediately after C. Since $b < c = a$, the place of size b immediately following A is now free, so white B can be put there. Finally, if $c > a$, the algorithm temporarily removes B[5], and calls the pebble at the closing position of size a by D (there must be a pebble present). The proof is concluded by considering two final sub-cases based on whether D is white or black. In the first case, the action is depicted on Figure 4. In case of black D the situation is as follows from Figure 5. □

3.2 Procedure DELETE

The deletion request requires to remove one pebble of a specified level. In our approach, the last (rightmost) pebble A of the requested level is chosen (see

[5] assume w.l.o.g. that $a \geq b$, see [5]

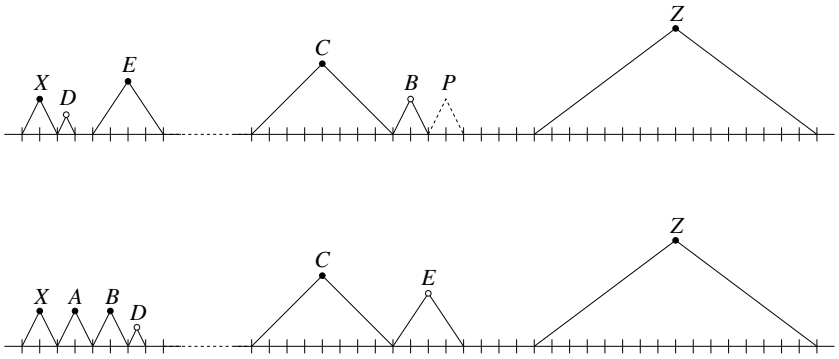

Fig. 4. An example of executing lines 26–31 of procedure INSERT

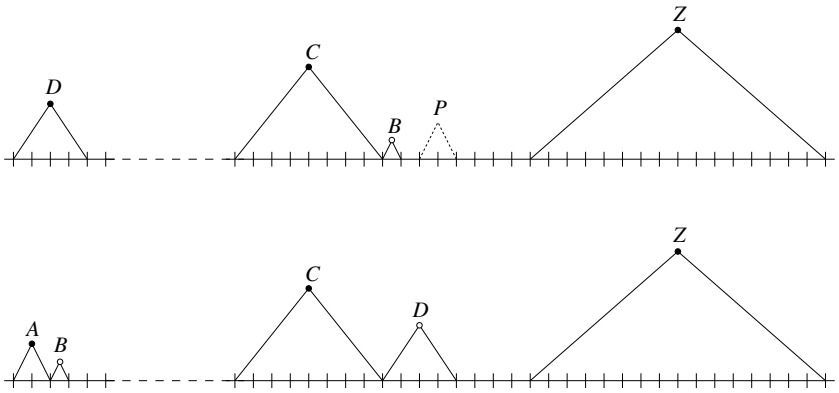

Fig. 5. An example of executing lines 33–35 of procedure INSERT

Procedure DELETE, Algorithm 2). However, this action may violate the invariants **P1**–**P3**. To remedy this, the algorithm uses several iterations to "push the problem" to the right. The main "problem" in this case is the free place caused by removing A. The "pushing" is done by selecting a suitable pebble X to the right of A, removing it, and using it to fill in the gap. A new iteration then starts to fix the problem at X's original place. The suitable candidate is found as follows: from all pebbles to the right of A, the smallest one is taken; if there are more pebbles of the smallest size available, the rightmost one is chosen. The correctness is proved by showing that this procedure is well defined, and that after a finite number of iterations, a valid situation is obtained (the full proof can be found in [5]):

Theorem 2. *Consider a valid situation, and a deletion request of size a. Procedure* DELETE *correctly processes the request, resulting in a valid situation.*

Sketch of proof. Let us number the iterations of the **while** loop by $t = 1, 2, \ldots$. Let the t-th iteration starts with the configuration of pebbles and free places Γ_t,

Algorithm 2 Procedure DELETE removes the last pebble of level l.

```
 1: let A be the last pebble of level l, and i be the starting position of A
 2: remove A
 3: while there are any pebbles to the right of i do
 4:     let x be the size of the smallest pebble to the right of i
 5:     if there is a free place of size x starting at i then
 6:         let X be the rightmost pebble of size x
 7:         let j be the starting position of X
 8:         move X to i
 9:         if X has white left neighbor Q, and Q has a left neighbor W of size x then
10:             swap X and Q                                  ▷ W is black, w ≤ x
11:         end if
12:         let i:=j
13:     else
14:         exit
15:     end if
16: end while
```

and a position i_t; it selects a pebble X_t of size x_t starting at j_t, moves it to i_t, swaps it with its white neighbor if necessary, and sets $i_{t+1} := j_t$.

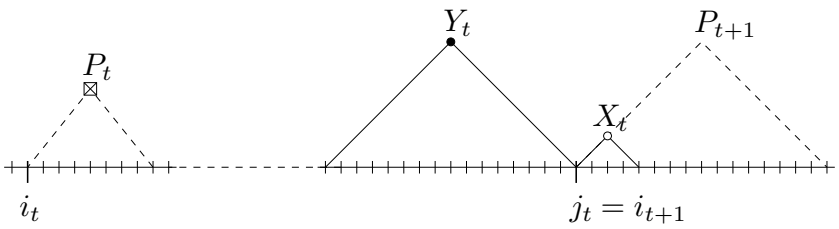

Fig. 6. One iteration of the **while** loop in procedure DELETE.

Denote by P_t the free place of size a_t starting at i_t such that $a_1 = a$, and the size a_{t+1} is determined as follows: let Γ'_t be obtained from Γ_t by putting a new pebble A_t on P_t. If the color of X_t in Γ'_t is black, then $a_{t+1} := x_t$, otherwise $a_{t+1} := y_t$, where Y_t is X_t's left neighbor. It can be shown that this definition is correct, i.e. that P_{t+1} is indeed free. To do so, the following claim will be show by induction on t:

For every iteration t of the **while** *loop, the corresponding Γ'_t is a valid situation*

For $t = 1$ the iteration starts just after A was removed from a valid situation Γ and replaced by A_1 with $a_1 = a$, so the basis of induction follows.

For the inductive step, consider the t-th iteration. The algorithm either stops or enters the next iteration. We prove that in the latter case Γ'_{t+1} is valid, provided Γ'_t was valid. We distinguish two cases:

– if $x_t < a_t$, it is possible to prove that there are neither white pebbles nor free places between i_t and j_t, from which the invariants readily follow;
– if $x_t \geq a_t$, then the swap on line 10 never happens, and it is again possible to argue the validity of the resulting situation.

A separate analysis of the last iteration concludes the proof of the theorem: the algorithm stops either when there are no pebbles to the right of i_t, or when the selected pebble X_t does not fit to position i_t. In both cases it is possible to show that the resulting situation is valid. □

4 Complexity

This section is devoted to the analysis of the average number of reassignments per request needed in the worst-case computation. Obviously, INSERT requires at most constant number of reallocations. The situation with DELETE is more complicated. We develop an accounting scheme that ensures a linear (in the number of requests) number of iterations of the **while** loop over the whole computation. To this end, we introduce the notion of *coins*: each request pays a constant number of coins to the accounting system, and each individual iteration of the **while** loop in DELETE spends one coin from the system.

Each request is associated with a constant number of coins which can be put on some places. We show that it is possible to put the coins in such a way that each iteration of the loop in DELETE can be paid by an existing coin. In our coin placement strategy we shall maintain the following additional invariant:

P4: Consider a free place P such that there are some pebbles to the right of P. Let X be the smallest pebble to the right of P. Then there are at least $\lfloor 2p/x \rfloor$ coins on P.

From now on we shall consider situations with some coins placed in some places, and we show how to manage the coins so that there is always enough cash to pay for each iteration in DELETE. The following two lemmas present the accounting strategy for INSERT and DELETE:

Lemma 2. *Let us suppose that procedure* INSERT *was called from a situation in which invariants* **P1**– **P4** *hold. Then it is possible to add a constant number of coins and reallocate the existing ones in such a way that invariants* **P1**– **P4** *remain valid.*

Proof. It has already been proven that invariants **P1**– **P3** are preserved by procedure INSERT, so it is sufficient to show how to add a constant number of coins in order to satisfy **P4**. During procedure INSERT, only a constant number of pebbles are *touched* – i.e. added or reassigned. Let Γ be the situation before INSERT and Γ' be the situation after INSERT finished. Let P be a free place in Γ' and X be the smallest pebble to the right of P. We distinguish two cases:

Case 1: X was touched during INSERT

If $p < x/2$, no pebbles are required on P, so let us suppose that $p \geq x/2$. However, since Γ' is valid, the free bandwidth before X is less than x, so $p = x/2$, and there is only one pebble required in P; this pebble will be placed on P and charged to X. Obviously, for each touched pebble X, there may be only one free place of size $x/2$ to the left (because of **P1**), so every touched pebble will be charged at most one coin using in total constant number of coins.

Case 2: X was not touched during INSERT
If P was free in Γ the required amount of $\lfloor 2p/x \rfloor$ coins was already present on P in Γ, so let us suppose that P was not free in Γ. That means that P became free in the course of INSERT when some pebbles were reallocated. Using similar arguments as in the previous case we argue that $p = x/2$. However, during INSERT only a constant number of free places of a given size could be created, so it is affordable to put one coin on each of them. □

Lemma 3. *Let us suppose that procedure* DELETE *was called from a situation in which invariants* **P1**– **P4** *hold. Then it is possible to add a constant number of coins, remove one coin per iteration of the main loop, and reallocate the remaining coins in such a way that invariants* **P1**– **P4** *remain valid.*

Sketch of proof. Let us suppose that there is at least one full iteration of the main loop. Recall the notation from the proof of Theorem 2, i.e. we number the iterations of the main loop, and Γ_t is the configuration at the beginning of t-th iteration. Moreover, Γ'_t is obtained from Γ_t by putting a new pebble A_t on P_t. It follows from the proof of the theorem that Γ'_t is always a valid situation. We use induction on t to show that the following can be maintained:

For every iteration $t > 1$ of the **while** *loop, the corresponding Γ'_t satisfies* **P1**– **P4**, *all pebbles to the right of i_t have size at least a_t, and one extra coin lays on A_t. Moreover, if some pebble to the right of i_t has the size a_t, then two extra coins lay on A_t.*

We omit the details about the induction basis, and proceed with the inductive step. In order to prove the claim for Γ'_{t+1}, consider the situation at the end of the t-th iteration when the algorithm enters the $(t+1)$-st one. Γ'_{t+1} is obtained from Γ'_t by removing A_t, moving X_t to i_t, and placing A_{t+1} on $i_{t+1} = j_t$. Note that in this case $x_t \geq a_t$, so there is no swap. It is possible to show that all pebbles to the right of i_{t+1} have size at least a_{t+1}, and that there is no free place between i_t and j_t in Γ'_{t+1}. Since for the free places before i_t and after j_t, **P4** remains valid, we argue that **P4** holds in Γ'_{t+1}.

Now we show how to find two free coins – one to pay for the current iteration, and one to be placed on A_{t+1}. If $x_t = a_t$, there are two coins placed on A_t. Otherwise (i.e. in case that $x_t > a_t$) one coin comes from the deletion of A_t and the other can be found as follows. Since X_t was placed on i_t, and $x_t > a_t$, there must have been a free place P of size $x_t/2$ in Γ'_t. Moreover, X_t was to the right of P, and so there must have been at least one coin on P. In Γ'_{t+1}, P is covered by X_t, so the coin can be used.

The last thing to show is to find a second free coin in case that there exists some pebble Q to the right of A_{t+1} of size $q = a_{t+1}$. In this case X_t is white in

Γ'_t – otherwise $a_{t+1} = x_t$ would hold and there would be no pebble of size x_t to the right of X_t. Hence $x_t \leq a_{t+1}/2$ and A_{t+1} in Γ'_{t+1} covers a place of size $a_{t+1}/2$ that has been free in Γ'_t. According to **P4** there is a coin on this place in Γ'_t; this coin can be used.

The proof of the theorem is concluded by considering the last iteration. Let $\Gamma'_{t_{fin}}$ be the last situation. The final situation is obtained from $\Gamma'_{t_{fin}}$ by removing $A_{t_{fin}}$. We show that **P4** holds. The only free place that could violate **P4** is the one remained after $A_{t_{fin}}$, however, there was a coin on $A_{t_{fin}}$, and all pebbles to the right (if any) are bigger than $a_{t_{fin}}$. □

5 Conclusion

We have presented an online algorithm for bandwidth allocation in wireless networks, which can be used to perform the OVSF code reallocation with the amortized complexity of $O(1)$ reallocations per request. This is an improvement over the previous best known result achieving the competitive ratio of $O(n)$. Moreover, the constant in our algorithm is small enough to be of practical relevance.

On the other hand, no attempt has been made at minimizing this constant. With the best known lower bound of 1.5 it would be worthwhile to close the gap even further.

References

1. F. Adachi, M. Sawahashi, and K. Okawa. Tree structured generation of orthogonal spreading codes with different lengths for the forward link of DS-CDMA mobile radio. *IEE Electronic Letters*, 33(1):27–28, 1997.
2. G. S. Brodal, E. D. Demaine, and J. I. Munro. Fast allocation and deallocation with an improved buddy system. *Acta Informatica*, 41(4–5):273–291, March 2005.
3. D. C. Defoe, S. R. Cholleti, and R. K. Cytron. Upper bound for defragmenting buddy heaps. In *LCTES '05: Proceedings of the 2005 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*, pages 222–229, New York, NY, USA, 2005. ACM Press.
4. T. Erlebach, R. Jacob, M. Mihalák, M. Nunkesser, G. Szabó, and P. Widmayer. An algorithmic view on OVSF code assignment. In V. Diekert and M. Habib, editors, *STACS*, volume 2996 of *Lecture Notes in Computer Science*, pages 270–281. Springer, 2004.
5. M. Forišek, B. Katreniak, J. Katreniaková, R. Královič, R. Královič, V. Koutný, D. Pardubská, T. Plachetka, and B. Rovan. Online bandwidth allocation. *Technical report*, 2007, arXiv:cs/0701153v1.
6. K. C. Knowlton. A fast storage allocator. *Communications of ACM*, 8(10):623–624, 1965.
7. T. Minn and K.-Y. Siu. Dynamic assignment of orthogonal variable-spreading-factor codes in W-CDMA. *IEEE Journal on Selected Areas in Communications*, 18(8):1429–1440, 2000.
8. Wikipedia. Universal mobile telecommunications system. Available at: `http://en.wikipedia.org/wiki/Umts`. From Wikipedia, the free encyclopedia [Online; accessed 23. March 2006].