Online Tree Node Assignment with Resource Augmentation

Joseph Wun-Tat Chan[1], Francis Y.L. Chin[2,⋆], Hing-Fung Ting[2,⋆⋆], and Yong Zhang[2]

[1] Department of Computer Science, King's College London, London, UK
joseph.chan@kcl.ac.uk
[2] Department of Computer Science, The University of Hong Kong, Hong Kong
{chin,hfting,yzhang}@cs.hku.hk

Abstract. Given a complete binary tree of height h, the *online tree node assignment problem* is to serve a sequence of assignment/release requests, where an *assignment request*, with an integer parameter $0 \leq i \leq h$, is served by assigning a (tree) node at level (or height) i and a *release request* is served by releasing a specified assigned node. The node assignments have to guarantee that no node is assigned to two assignment requests unreleased, and every leaf-to-root path of the tree contains at most one assigned node. With assigned node reassignments allowed, the target of the problem is to minimize the number of assignments/reassigments, i.e., the cost, to serve the whole sequence of requests. This online tree node assignment problem is fundamental to many applications, including OVSF code assignment in WCDMA networks, buddy memory allocation and hypercube subcube allocation.

Most of the previous results focus on how to achieve good performance when the same amount of resource is given to both the online and the optimal offline algorithms, i.e., one tree. In this paper, we focus on resource augmentation, where the online algorithm is allowed to use more trees than the optimal offline algorithm. By using different approaches, we give (1) a 1-competitive online algorithm, which uses $(h+1)/2$ trees, and is optimal because $(h+1)/2$ trees are required by any online algorithm to match the cost of the optimal offline algorithm with one tree; (2) a 2-competitive algorithm with $3h/8+2$ trees; (3) an amortized $(4/3+\alpha)$-competitive algorithm with $(11/4+4/(3\alpha))$ trees, for any α where $0 < \alpha \leq 4/3$.

1 Introduction

The tree node assignment problem is defined as follows. Given a complete binary tree of height h, the target is to serve a sequence of requests. Every request is classified as either an *assignment request* or a *release request*. To serve an assignment request, which is associated with an integer parameter $0 \leq i \leq h$, we

⋆ Supported by HK RGC grant HKU-7113/07E.
⋆⋆ Supported by HK RGC grant HKU-7171/08E.

H.Q. Ngo (Ed.): COCOON 2009, LNCS 5609, pp. 358–367, 2009.

have to assign it a (tree) node at level (or height) i. To serve a release request, we just need to mark a specified assigned node free. There are two constraints for the node assignments, which are (1) any node can be assigned to at most one unreleased assignment request, (without ambiguity, all assigned requests are assumed unreleased), and (2) there is at most one assigned node in every leaf-to-root path. Fig. 1 gives a valid tree node assignment.

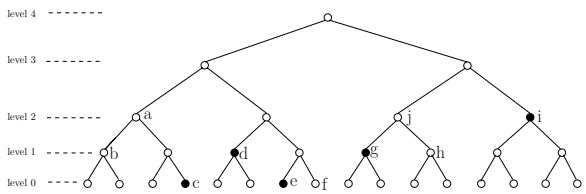

Fig. 1. Example of a valid nodes assignment. Filled circles represent assigned nodes.

The tree node assignment problem can be considered as a general resource allocation problem, which can model the specific problems, such as the Orthogonal Variable Spreading Factor (OVSF) code assignment problem [2, 3, 7, 12, 13, 14, 15,16,17], the buddy memory allocation problem [1,5,10,11], and the hypercube subcube allocation problem [6]. The main difference between these problems is how the resource, the nodes at level i, for $0 \leq i \leq h$, are interpreted. In the OVSF code assignment problem, the resource consists of codes of frequency bandwidth 2^i; in the buddy memory allocation problem, the resource consists of memory blocks of size 2^i; in the hypercube subcube allocation problem, the resource consists of subcubes of 2^i processors.

Similar to the memory allocation problem, algorithms for the tree node assignment problem also face the *fragmentation problem*. For example, in Fig. 1, there is no node of level 2 that can be assigned (without violating constraint (2) of node assignments). In fact, we can "defragment" the tree by reassigning the assigned node c to free node f. Then, node a is free to assign to assignment request of level 2. In this paper, we consider the tree node assignment problem where *reassignment* of nodes is allowed. In addition, we design algorithms that serve *all* requests in the request sequence, and we assume that all requests in the request sequence can be served by some algorithm using only one tree of height h.

The performance of an algorithm for the tree node assignment problem is measured by the number assignments/reassignments, which is called the *cost*, carried out by the algorithm. The release operations take no cost, as in the applications, the operation to release a resource is usually negligible when compared with the overhead of assigning/reassigning a resource.

Both the offline and online version of the tree node assignment problem are well studied, especially in the context of OVSF code assignment. In the offline version, the sequence of requests is known in advance, whereas in the online version, the algorithm must process each request without any information about future requests. The offline version of this problem is proved to be NP-hard [16].

Most of the previous work studied the online version of the problem, where performance is given in terms of competitive ratios, i.e., the worst case ratio of the costs between the online algorithm and the optimal offline algorithm. Erlebach et al [7] gave an $O(h)$-competitive algorithm, where h is the height of the tree, and proved a general lower bound on the competitive ratio of at least 1.5. Forisek et al [8] gave a constant-competitive algorithm, but without deriving the exact value of the constant. Chin, Ting and Zhang [2] gave a 10-competitive algorithm and, in addition, their algorithm guarantees that each request is served with at most 5 assignments/reassignments. They then improved the upper bound by proposing a 6-competitive algorithm [3]. They improved the lower bound of the competitive ratio to $5/3 \approx 1.67$ [2]. Very recently, Miyazaki and Okamoto [14] gave a 7-competitive algorithm and improved the lower bound of the competitive ratio to 2.

In this paper, we focus on the online version of the problem with resource augmentation [9], which means that the online algorithm is allowed to use more trees than the optimal offline algorithm. We assume that the optimal offline algorithm uses one tree, while the online algorithm can use k trees, where $k \geq 1$. The competitive ratio is defined to be the worst case ratio of the cost between the online algorithm with k trees and the optimal offline algorithm with one tree. This problem has been studied before. Erlebach et al [7] gave a 4-competitive algorithm with two trees, and Chin, Zhang and Zhu [4] gave a 5-competitive algorithm with 9/8 trees.

The main contribution of this paper is to show how the competitive ratio can be further reduced by making use of more trees. In other words, how the future information (offline) can be compensated by extra resources (trees). First, we give an online algorithm with $(h+1)/2$ trees that matches the cost of the optimal offline algorithm with one tree. In fact, this algorithm even matches the cost of each request with that of the optimal offline algorithm with one tree, as it does not require any reassignments. We further show that for any online algorithm to match the cost of the optimal offline algorithm with one tree, $(h+1)/2$ trees are necessary. That implies that our 1-competitive algorithm is optimal in terms of the number of trees used. Then, by using one extra reassignment for each release request to reduce the fragmentation of the assigned nodes in the tree, we can use fewer trees to serve the sequence of requests. In particular, we give a 2-competitive online algorithm with $3h/8 + 2$ trees. This algorithm bounds the cost of each request to one, i.e., each assignment request takes one assignment and each release request takes at most one reassignment. These two algorithms with bounded costs for each request are presented in Section 2.

When it is not necessary to bound the cost of individual requests to a constant, we can achieve an amortized $(4/3+\alpha)$-competitive algorithm with $(11/4+4/(3\alpha))$ trees, for any α where $0 < \alpha \leq 4/3$. This algorithm is presented in Section 3.

Remark: Because of page limit, some detailed proofs are removed. For full version of this paper, please refer to `http://www.cs.hku.hk/$\sim$yzhang/tree.pdf`

2 Algorithms with Bounded Cost per Request

We present two algorithms in this section. For any request sequence σ where the optimal algorithm can satisfy the requests with one tree,

- the first algorithm uses at most $(h+1)/2$ trees and incurs a cost of one for each assignment request and a cost of zero for each release request;
- the second algorithm uses at most $3h/8+2$ trees and incurs a cost of at most one for each assignment and release request.

Since any algorithm needs to assign a node for each assignment request, the first algorithm is optimal in terms of the cost, i.e., 1-competitive. We further show that it is necessary for any online algorithm to use $(h+1)/2$ trees in order to match the cost of the optimal algorithm with one tree. Since the number of release requests is at most the number of assignment requests, the total cost incurred by the second algorithm is at most twice that of the optimal algorithm with one tree, i.e., 2-competitive.

2.1 Preliminary

We introduce some definitions in this part, which are used in subsequent sections. A node v is called a *free node* if there is no assigned node in any leaf-to-root path going through the node v. A node v is *blocked* if there is a path from the root to a leaf through the node v that contains an assigned node. Node v is also called a *blocked node*, moreover, we say node v is *blocked by an assigned node at some level*. A node at level i or an assignment request that asks for a node at level i is said to have a *bandwidth* of 2^i. It is clear that to serve a set of assignment requests or to accommodate a set of assigned nodes in a tree of height h, the total bandwidth of the requests or nodes has to be no more than 2^h.

2.2 Optimal 1-Competitive Algorithm

The idea to achieve the optimal cost is to dedicate some subtrees to serve assignment requests of some particular levels. To describe this assignment scheme, we define a *half-tree* to be the subtree rooted at either the left or right child of a root. This online algorithm uses $h+1$ half-trees. We label the $h+1$ half-trees from 0 to h. When there is a level-i assignment request with $i < h$, we pick from half-trees 0 to $i+1$ any free node at level i and assign it to the request. If $i = h$, we assign the root of any tree to the request, as there should be no other assigned nodes. For any release request, we just release the assigned node and mark it free.

The correctness of the algorithm depends on whether, for each level-i assignment request, we can always find a level-i free node from the half-trees 0 to $i+1$. The following lemma makes sure that it can be done.

Lemma 1. *If the total bandwidth of the assigned nodes and the new-coming assignment request of level i is less than 2^h, there is always a free node at level i in the half-trees 0 to $i+1$ for $i < h$.*

Theorem 2. *For the online tree node assignment problem, we have an 1-competitive algorithm using* $(h+1)/2$ *trees, where the cost of serving each assignment request is one and the cost of serving each release request is zero.*

Lower bound of number of trees to achieve 1-competitiveness. We give an adversary such that the optimal algorithm with one tree serves each assignment request with only one assignment and each release request with no reassignment. At the same time, for any online algorithm that wants to limit the cost as the optimal algorithm, the adversary forces it to use at least $(h+1)/2$ trees.

The main idea of the adversary is to send assignment requests in ascending order of their levels. The adversary then releases some requests but makes sure that the remaining assigned nodes of low level block a significant part of the trees. Thus, the assignment requests of high level need to be served with extra trees. The adversary is divided into h steps, where in Step i, assignment requests of level i are sent, and then some release requests of level $j \leq i$ follow, except for Step $h-1$. Over all time, the total bandwidth of the assigned nodes is kept at most 2^h. The details of the adversary is given as follows.

Step 0: The adversary sends 2^h level-0 assignment requests. Any online algorithm must assign 2^h level-0 free nodes. The adversary then releases 2^{h-1} of the level-0 assigned nodes such that $2^{h-1} = 2 \cdot 2^{h-2}$ level-1 nodes are blocked.

Step 1: The adversary sends 2^{h-2} level-1 assignment requests. After the online algorithm has assigned 2^{h-2} level-1 free nodes, the adversary releases 2^{h-2} level-0 and 2^{h-3} level-1 assigned nodes, i.e., half of the assigned nodes at each level with assigned nodes. The release requests make sure that it results in $3 \cdot 2^{h-3}$ nodes blocked at level-2.

...

Step i**, for** $2 \leq i \leq h-2$**:** The adversary sends 2^{h-i-1} level-i assignment requests. After the online algorithm has assigned 2^{h-i-1} level-i free nodes, the adversary releases 2^{h-i-1} level-0 assigned nodes and 2^{h-i-2} level-j assigned nodes for $1 \leq j \leq i$, i.e., half of the assigned nodes at each level with assigned nodes. The release requests make sure that it results in $(i+2) \cdot 2^{h-i-2}$ nodes blocked at level-$(i+1)$.

...

Step $h-1$**:** The adversary sends one level-$(h-1)$ assignment requests. (Now, there are $h+1$ nodes blocked at level $h-1$.)

Lemma 3. *For any online algorithm, at the end of Step* i*, the number of level-*$(i+1)$ *nodes blocked is* $(i+2) \cdot 2^{h-i-2}$*, for* $0 \leq i \leq h-2$*.*

It is easy to construct an offline algorithm with one tree so that it serves for the adversary each assignment request with one assignment and each release request with no reassignment. Thus, we have the following theorem.

Theorem 4. *No online algorithm can match the cost of the optimal offline algorithm with one tree by using less than* $(h+1)/2$ *trees.*

2.3 2-Competitive Algorithm with Bound Cost per Request

This section gives an online algorithm that uses fewer trees, i.e., $3h/8 + 2$, but comes with a slightly higher competitive ratio, i.e., 2. The main idea of the algorithm is to apply an extra reassignment for any release request to reduce fragmentation of the tree (to reduce blocking bandwidth) by pairing two assigned nodes with unassigned siblings. Precisely, the algorithm serves each assignment and release request with at most one assignment or reassignment. As the number of release requests is at most the number of assignment requests, the total cost of the algorithm is at most twice that of the optimal algorithm.

We design an assignment scheme for the $3h/8+2$ trees available to the online algorithm. First, we define an *eighth-tree* to be a subtree rooted at a level-$(h-3)$ node. All the $3h$ eighth-trees in the $3h/8$ trees are labeled. Six of them are labeled 0 and three of them are labeled i, for $1 \leq i \leq h-2$. Denote the other two trees as T and T^*. In general, the $3h$ eighth-trees are to handle the assignment requests at level-i for $0 \leq i \leq h-3$, T for level-$(h-2)$ and -$(h-1)$, and T^* for all levels. In particular, at any time, we allow at most one assigned node in each level $0 \leq i \leq h$ of T^*. It enables us to find a free node at level-i of T^*, whenever there is no assigned node at level-i.

The details of the assignment scheme are given as follows.

Assignment request R of level i:

If there is no level-i assigned node in T^*, assign R a level-i free node in T^*. Otherwise,

- If $i = h-2$ or $i = h-1$, assign R any level-i free node in T.
- If $0 \leq i \leq h-3$, assign R any level-i free node from any eighth-tree labeled k for $0 \leq k \leq i$. If no free node is available, consider the eighth-trees labeled $i+1$. If there is a level-i free node v where v's sibling is an assigned (level-i) node, assign R the free node v. Otherwise, assign R any level-i free node in any eighth-tree labeled $i+1$. (Lemma 4 shows that an level-i free node always exists in one of the eight-tree with label from 0 to $i+1$).

Release request R of level i:

Release the node assigned to R and mark it free.

If $0 \leq i \leq h-3$, consider the following situations for reassignment.

- If there is no assigned node at level i of T^* and there is a level-i assigned node v in an eighth-tree labeled $i+1$ where v's sibling is a free (level-i) node, reassign v to a level-i free node of T^*.
- If there are two level-i assigned node u and v in an eighth-trees labeled $i+1$ where both u's and v's siblings are a free (level-i) node, reassign u to v's sibling.

To ensure the correctness of the algorithm, we show that the followings are true.

1. When T^* contains an assigned node at level i, for $i = h-1$ or $i = h-2$, there is always a level-i free node in T. For this case, it is clear as otherwise, the total bandwidth of the assigned nodes is at least 2^h.

2. When T^* contains an assigned node at some level i ($0 \leq i \leq h-3$), there is always a level-i free node in some eighth-tree labeled k, for $0 \leq k \leq i+1$.

In order to ensure that the Property (2) is true (by Lemma 6), we may spend an extra reassignment after a release request to tidy up the configuration of the assigned nodes in the eighth-trees. We want to maintain a configuration of the assigned nodes in the eighth-trees as in the following lemma.

Lemma 5. *Let $0 \leq i \leq h-3$. (1) When T^* contains no assigned node at level i, there is no assigned node v at level i of any eight-tree labeled $i+1$ where v's sibling is a free node. (2) If there exist one assigned node v, among all assigned nodes, at level i of some eight-tree labeled $i+1$ where v's sibling is a free node, T^* must contains an assigned node at level i.*

Lemma 6. *Assume that the total bandwidth of the assigned nodes is less than 2^h. For any i, $0 \leq i \leq h-3$, when T^* contains an assigned node at level i , there is always a level-i free node in some eighth-tree labeled k, for $0 \leq k \leq i+1$.*

Theorem 7. *For the online tree node assignment problem, we have a 2-competitive algorithm using $3h/8+2$ trees and the cost of serving each request is at most one.*

3 Algorithm with Amortized Constant Cost per Request

In this section, we give a $(4/3+\alpha)$-competitive algorithm using $11/4+(4/(3\alpha)$ trees, for any α where $0 < \alpha \leq 4/3$. This algorithm is based on an extended concept of *compact configuration* of assigned nodes in trees [7], which is described below.

Assume that the available trees to the online algorithm are arranged on a line. For any two nodes u and v, where u is not an ancestor of v and vice versa, we say that u is on the *left* of v if u is in a tree which is on the left of the tree containing v, or there is a leaf in the subtree rooted at u which is on the left of a leaf in the subtree rooted at v; otherwise, u is on the *right* of v. Level i of the trees is defined to be *compact* if all nodes of level i to the left of a blocked node of level i, which is blocked by an assigned node at level no more than i, are also blocked. We say that a configuration is compact if all levels of the trees are compact.

It is very costly to maintain a compact configuration after serving each request. By using a "relaxed compact" configuration with a constant number of trees, the amortized competitive ratio can be reduced to a constant. The idea of the relaxation is as follows: for each level i, there may exist more free nodes than the compact configuration. Such kind of free nodes can accommodate the following assignment request immediately without reassigning nodes at higher levels. Thus, there may be no extra cost (reassignment) after serving some request and the amortization of each request can be reduced to a constant by using some extra resources (the free node to the right of some assigned nodes may be wasted).

To improve the competitive ratio, we could be lazy in tidying up the configuration when serving assignment or release requests. In this part, we define a less "tidy" configuration called the *semi-compact configuration*, which stores odd-level and even-level assigned nodes separately. However, the way to store the two sets of assigned nodes is the same. For simplicity, we would show how even-level assigned nodes are stored only.

To describe the algorithm, we define a notation called the *level-i region*, which consists of all level-i assigned nodes and maybe some level-i free nodes. There is a level-i region for each level i. If there is no assigned nodes at level i, the level-i region may consist of only free level-i nodes or an empty region.

The semi-compact configuration (as shown in Figure 2) divides the level-i region into two contiguous parts, the *main region* on the left and the *gap region* on the right.

- The main region consists of assigned nodes and maybe some free nodes, but the number of free nodes is at most β times the number of assigned nodes, where $\beta \geq 1$ is a fixed parameter.
- The gap region consists of only free nodes and the number of free nodes is at most 7.

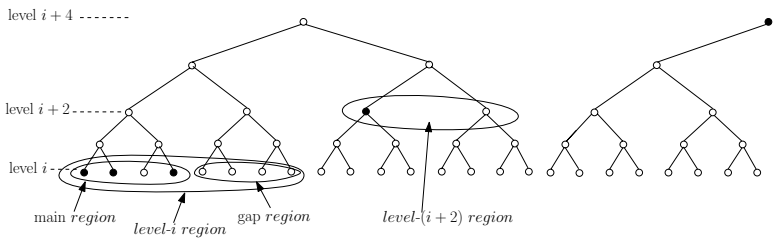

Fig. 2. An example of an semi-compact configuration

The details of the algorithm is given as follows.

Assignment request R of level i:

Case A1. If there is a level-i free node in the level-i main region, assign R the free node.

Case A2. If there is a level-i free node in the level-i gap region, assign R the leftmost free node, say u, and u is moved from the gap to the main region.

Case A3. If there is no level-i free node in the level-i region, find the non-empty level-j region G_j for the smallest $j > i$

(1) If G_j has no assigned node, i.e., the leftmost node is a free node, the free node is "divided" into four level-i free nodes, three level-k nodes for even-level k between $i+2$ to $j-2$. These free nodes are inserted to the corresponding level-i and level-k gap regions. The leftmost level-i free node is assigned to R. Similar to the case A2, the newly assigned node is moved from the gap to the main region.

(2) Otherwise, release the leftmost assigned node, say u, of G_j, which is reassigned later. The released node is divided and assigned as in step (1). An assignment request of level j is issued to find a free node for u.

Release request R of level i:

Release the assigned node for R.

Case R1. If the number of free nodes in the level-i main region is at most β times the number of assigned nodes, do nothing.

Case R2. If the number of free nodes in the level-i main region is more than β times the number of assigned nodes, compact the main region into contiguous assigned nodes on the left by reassignments. The free nodes are moved to the gap regions.

(3) If the number of free nodes in the gap region is at most 7, do nothing.

(4) If the number of free nodes in the gap region is more than 7, let the number be in the form $4x + y$ where x and y are integers and $x > 1$ and $4 \leq y \leq 7$. Group the rightmost $4x$ free nodes into x level-$(i+2)$ free nodes. The x level-$(i+2)$ free nodes are moved to the level-$(i+2)$ main region in a way that are considered as x release requests.

We use an amortized analysis to bound the average number of assignments and reassignments needed for serving each assignment or release request. The credits paid for an assignment request is $4/3$ and a release request is α where $\alpha = 4/(3\beta) \leq 4/3$ as $\beta \geq 1$. The potential of a level-i main region is α times the number of free nodes in the main region. The potential of the level-i gap region is defined as follows.

Number of free nodes in the gap region	0	1	2	3	4	5	6	7
Potential	1	2/3	1/3	0	0	$\alpha/4$	$\alpha/2$	$3\alpha/4$

The potential of a semi-compact configuration is the sum of all potential of the level-i main and gap regions for all level i. The initial semi-compact configuration has four level-0 free nodes in the level-0 gap region, and three level-i free nodes in the level-i gap region for all other $i > 0$. All main regions are empty. The initial potential is 0.

The following lemma shows that the saved credit is able to pay the actual cost of the algorithm for serving each request.

Lemma 8. *Let S_b and S_a be the potential of the semi-compact configuration before and after serving a request. The actual cost to serve an assignment request is at most $4/3 - (S_a - S_b)$ and a release request is at most $\alpha - (S_a - S_b)$.*

Summing up all node trees used, we can prove the following theorem.

Theorem 9. *For the online tree node assignment problem, our algorithm in this section is $(4/3 + \alpha)$-competitive and it uses at most $11/4 + 4/(3\alpha)$ trees, for any α where $0 < \alpha \leq 4/3$.*

References

1. Brodal, G.S., Demaine, E.D., Munro, J.I.: Fast allocation and deallocation with an improved buddy system. Acta Inf. 41(4-5), 273–291 (2005)
2. Chin, F.Y.L., Ting, H.F., Zhang, Y.: A constant-competitive algorithm for online OVSF code assignment. In: Tokuyama, T. (ed.) ISAAC 2007. LNCS, vol. 4835, pp. 452–463. Springer, Heidelberg (2007)
3. Chin, F.Y.L., Ting, H.F., Zhang, Y.: Constant-Competitive Tree Node Assignment (manuscript)
4. Chin, F.Y.L., Zhang, Y., Zhu, H.: Online OVSF code assignment with resource augmentation. In: Kao, M.-Y., Li, X.-Y. (eds.) AAIM 2007. LNCS, vol. 4508, pp. 191–200. Springer, Heidelberg (2007)
5. Defoe, D.C., Cholleti, S.R., Cytron, R.: Upper bound for defragmenting buddy heaps. In: Proceedings of the 2005 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems, pp. 222–229 (2005)
6. Dutt, S., Hayes, J.P.: Subcube allocation in hypercube computers. IEEE Trans. Computers 40(3), 341–352 (1991)
7. Erlebach, T., Jacob, R., Mihalák, M., Nunkesser, M., Szabó, G., Widmayer, P.: An algorithmic view on OVSF code assignment. Algorithmica 47(3), 269–298 (2007)
8. Forišek, M., Katreniak, B., Katreniaková, J., Královič, R., Královič, R., Koutný, V., Pardubská, D., Plachetka, T., Rovan, B.: Online bandwidth allocation. In: Arge, L., Hoffmann, M., Welzl, E. (eds.) ESA 2007. LNCS, vol. 4698, pp. 546–557. Springer, Heidelberg (2007)
9. Kalyanasundaram, B., Pruhs, K.: Speed is as powerful as clairvoyance. J. ACM 47(4), 617–643 (2000)
10. Knowlton, K.C.: A fast storage allocator. Commun. ACM 8(10), 623–624 (1965)
11. Knuth, D.E.: The Art of Computer Programming. Fundamental Algorithms, vol. 1. Addison-Wesley, Reading (1975)
12. Li, X.-Y., Wan, P.-J.: Theoretically good distributed CDMA/OVSF code assignment for wireless ad hoc networks. In: Wang, L. (ed.) COCOON 2005. LNCS, vol. 3595, pp. 126–135. Springer, Heidelberg (2005)
13. Minn, T., Siu, K.-Y.: Dynamic assignment of orthogonal variable-spreading-factor codes in W-CDMA. IEEE Journal on Selected Areas in Communications 18(8), 1429–1440 (2000)
14. Miyazaki, S., Okamoto, K.: Improving the competitive ratio of the online OVSF code assignment problem. In: Proceedings of the 19th International Symposium on Algorithms and Computation (ISAAC), pp. 64–76 (2008)
15. Rouskas, A.N., Skoutas, D.N.: OVSF codes assignment and reassignment at the forward link of W-CDMA 3G systems. In: Proceedings of the 13th IEEE International Symposium on Peronal, Indoor and Mobile Radio Communications, vol. 5, pp. 2404–2408 (2002)
16. Erlebach, T., Jacob, R., Tomamichel, M.: Algorithmische Aspekte von OVSF Code Assignment mit Schwerpunkt auf Offline Code Assignment. Student thesis at ETH Zürich
17. Wan, P.-J., Li, X.-Y., Frieder, O.: OVSF-CDMA code assignment in wireless ad hoc networks. Algorithmica 49(4), 264–285 (2007)