Theoretical Computer Science ••• (••••) •••–•••

Contents lists available at SciVerse ScienceDirect

Theoretical Computer Science

www.elsevier.com/locate/tcs

Constant-competitive tree node assignment

Yong Zhang [a,b,*,1], Francis Y.L. Chin [b,2], Hing-Fung Ting [b,3]

[a] College of Mathematics and Computer Science, Hebei University, China
[b] Department of Computer Science, The University of Hong Kong, Hong Kong

ARTICLE INFO

Article history:
Received 15 September 2010
Received in revised form 1 March 2013
Accepted 8 May 2013
Communicated by T. Erlebach

Keywords:
Online algorithms
Tree node assignment
Competitive ratio

ABSTRACT

In this paper, we study the online tree node assignment problem, which is a generalization of the well studied OVSF code assignment problem. Assigned nodes in a complete binary tree must follow the rule that each leaf-to-root path must contain at most one assigned node. At times, it is necessary to swap assigned nodes with unassigned nodes in order to accommodate some new node assignment. The target of this problem is to minimize the number of swaps in satisfying a sequence of node assignments and releases.

This problem is fundamental, not only to the OVSF code assignment, but also to other applications, such as buddy memory allocation and hypercube subcube allocation. All the previous solutions to this problem are based on a sorted and compact configuration by assigning the nodes linearly and level by level, ignoring the intrinsic tree property in their assignments.

Our contributions are: (1) give the concept of safe assignment, which is proved to be unique for any fixed set of node-assignment requests; (2) an 8-competitive algorithm by holding the safe assignment; and (3) an improved 6-competitive variant of this algorithm. Our algorithms are simple and easy to implement and our contributions represent meaningful improvements over recent results.

1. Introduction

The problem we study in this paper involves the assignment and release of nodes in a complete binary tree when faced with a sequence of requests for either an assignment of a node at specified level of the tree, or a release of an assigned node of the tree. The only rule which dictates the assignments/releases is that the tree remains in a *legal configuration* where no two assigned nodes lie on a single path from the root to a leaf. Fig. 1 is an example of a legal tree with assigned nodes darkened and marked as c, d, e, g and i.

The requirement of our problem is to accommodate each request if at all possible and, in order to accommodate a new request, it might be necessary to *swap* nodes to "make room" for the new request, where *swap* means changing the position of an assigned node with an unassigned node at the same level. For example, in Fig. 1, to accommodate a request for node assignment at level 2, we could first change the assignment by swapping node c with node f and then satisfy the request with the assignment of node a at level 2. Alternatively, we could swap node g and node b and then assign node j at level 2 to satisfy the request. Since each swap represents processing overhead, we have the optimization objective of designing algorithms to solve this tree assignment problem so as to minimize the number of swaps.

* Corresponding author at: Department of Computer Science, The University of Hong Kong, Hong Kong.
E-mail addresses: yzhang@cs.hku.hk (Y. Zhang), chin@cs.hku.hk (F.Y.L. Chin), hfting@cs.hku.hk (H.-F. Ting).

[1] Research supported by NSFC (No. 11171086) and HKU Small Project Funding 7176218.
[2] Research supported by HK RGC grant HKU-711709E.
[3] Research supported by HKU Small Project Funding 7176115.

0304-3975/$ – see front matter
http://dx.doi.org/10.1016/j.tcs.2013.05.014

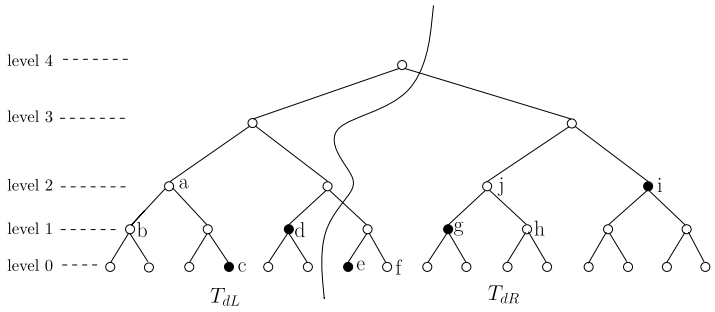


Fig. 1. Example of legal configuration, solid circles are assigned nodes.

Table 1
Application-specific problems and how the node and the swap are interpreted.

Problem	Node at level l	Swap
OVSF code assignment	Code of frequency bandwidth 2^l	Code reassignment
Buddy memory allocation	Memory block of size 2^l	Memory reallocation
Hypercube subcube allocation	Subcube of 2^l processors	Subcube migration

As it turns out, a solution to our problem, given its fundamental formulation, can be used to solve a variety of application-specific problems including the Orthogonal Variable Spreading Factor (OVSF) code assignment problem [2,4,8,11,14], the buddy memory allocation problem [1,6,12,13] and the hypercube subcube allocation problem [7]. The main difference between these problems is how the node at level l and the swap operation are interpreted (see Table 1).

As with other scheduling-type problems, there naturally arises an off-line and an online version of this problem. In the off-line version, the sequence of requests is made known to the algorithm at the outset, whereas in the online version, the algorithm must process each request without the benefit of any information about future requests. The off-line version of the tree node assignment problem is NP-hard [9], so the approach is to produce a good heuristic algorithm, whose performance is measured by the ratio of the cost of the heuristic algorithm to the cost of an optimal off-line algorithm. The ratio in case of the online problem (the off-line problem) is called the *competitive ratio* (*approximation ratio*).

There is a variety of ways to define the cost of an algorithm. For example, cost could simply be the total number of assignments and swaps done by the algorithm, with zero cost for each node release. This cost definition is justified since, for most applications, it is expensive to set up a node assignment while there is little cost associated with a node release (e.g. terminating communications, freeing some memory, or releasing the hypercube subcube for another task) and a swap can be considered equivalent to a node assignment plus a node release. In the actual application, higher level node assignment/swap for buddy memory allocation and hypercube subcube allocation might be more costly because larger size of the memory blocks and subcube are involved. However, a uniform cost for each level node assignment/swap is justified if the data transfer cost is considered to be negligible when compared with the high setup cost.

In this paper, we focus on the online problem and we define cost to be the total number of assignments and swaps done by the algorithm. For the online problem, Erlebach et al. [8] 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. With resource augmentation in the form of using a bigger tree, it is possible to have constant-competitive algorithms. Erlebach et al. [8] first gave a 4-competitive algorithm with two trees, Chin et al. [5] gave a 5-competitive algorithm with 9/8 trees. By balancing the performance ratio and resource augmentation, Chan et al. [3] gave a 2-competitive algorithm with $3h/8+2$ trees; a 8/3-competitive algorithm with 11/4 trees; and a general $(4/3+\alpha)$-competitive algorithm with $(11/4+4/(3\alpha))$ trees, for any $0<\alpha\leqslant 4/3$. Without resource augmentation, Forisek et al. [10] first gave a constant-competitive algorithm, but without deriving the exact value of the constant. According to their scheme, each node release may still cost $O(h)$ swaps, but with constant value potential defined on some nodes, a constant-competitive algorithm could be achieved. A 10-competitive algorithm [4] was also derived based on a lazy approach to group assigned nodes at some consecutive levels together. Miyazaki and Okamoto [14] introduced a 7-competitive algorithm and proved a lower bound of 2 on the competitive ratio.

All of the previous work has been based on a sorted and compact legal configuration, where all assigned nodes are sorted according to their levels in non-decreasing order from left to right in the tree and are pushed to the left as much as possible. In these methods, nodes are "packed" level by level, ignoring the intrinsic tree property in their assignments. By studying properties for a "good" legal configuration, one of our main contributions is to define a novel configuration, called *dense* configurations (to be defined in Section 2) which is not only a generalization of the sorted and compact configuration, but can fully utilize the capacity of the tree and allow new requests to be served easily. In fact, the sorted and compact configuration is one of two extreme cases of the dense configurations that require swaps in order to maintain the "dense" property. In this paper, we define a *safe* configuration, which is dense and easy to maintain, and does not have the problem of high cost in the sorted and compact configuration.

Please cite this article in press as: Y. Zhang et al., Constant-competitive tree node assignment, Theoretical Computer Science (2013), http://dx.doi.org/10.1016/j.tcs.2013.05.014

We give a simple algorithm (Section 3) in which, for each node assignment or release, we maintain a safe configuration with cost of at most 4. Thus, an 8-competitive algorithm can be achieved. After studying the worst case for the 8-competitive algorithm, we adopt a lazy-release approach, i.e. doing nothing for node releases until necessary, and this gives a 6-competitive scheme, which is the current best algorithm for the tree node assignment problem (Section 4).

2. Preliminaries

Consider any complete binary tree T. Let σ be its root and h be its height. T has 2^h leaves and σ is at level h. We assume that T is ordered and the nodes at a particular level are ordered from left to right. For any node u, let $T(u)$ be the subtree of T rooted at u, and $\mathtt{lv}(u)$ be the level of u. For any level $l > \mathtt{lv}(u)$, let $\mathtt{anc}_l(u)$ denote the level-l ancestor of u. Given any pair of nodes u, v that does not have any ancestor/descendant relationship, we say that u is to the left (right) of v if u's leaf descendants are to the left (right) of v's. For example, in Fig. 1, $\mathtt{lv}(c) = 0$, $\mathtt{anc}_2(c) = a$, the node f is to the right of b, and is to the left of i.

An *assignment* of T is the set of all assigned nodes in T. Throughout this paper, we will only consider assignments with legal configurations, i.e., no two assigned nodes lie on a single path from the root to a leaf. Each assigned node at level l would then "capture" a distinct set of 2^l leaves, which are its descendants. This motivates us to define the *demand* of a level-l node-assignment request to be 2^l. We have the following fact.

Fact 1. If the total demand of a set of node-assignment requests is more than 2^h, then no assignment can satisfy all these requests.

We say that node u is *free* with respect to an assignment A if none of its descendants and ancestors (including itself) are assigned nodes in A. For any set of nodes U, the *free capacity* $\mathtt{fc}(U)$ of U is defined to be the number of free leaf-descendants of nodes in U. We say that a level l is *dense* with respect to A if there is no free level-l node to the left of an assigned level-l node in A. We say that A is *dense* if all levels of T are dense with respect to A. The following lemma reveals a good property about dense assignments. In its proof, we use $\mathit{Left}(u)$ to denote all the level-$\mathtt{lv}(u)$ nodes to the left of u.

Lemma 1. *Suppose assignment A is dense. Consider any level l, let $u_0, u_1, \ldots, u_m$ be the sequence of nodes (from left to right) at this level. Then, for any $0 \leqslant i \leqslant m$, if there is no free node in $\{u_0, u_1, \ldots, u_i\}$, then $\mathtt{fc}(\{u_0, u_1, \ldots, u_i\}) < 2^l$.*

Proof. We prove by induction on the level l. The lemma is obviously true for $l = 0$. Assume that the lemma is true for levels $0, 1, 2, \ldots, l-1$, and we consider the nodes $u_0, u_1, \ldots, u_m$ at level l. Suppose that there is no free node in $\{u_0, u_1, \ldots, u_i\}$. By definition, these nodes are assigned or have assigned ancestors or assigned descendants. If all of them are assigned or have assigned ancestors, then $\mathtt{fc}(\{u_0, u_1, \ldots, u_i\}) = 0$ (because if a node x has an assigned ancestor, then $\mathtt{fc}(\{x\}) = 0$). Otherwise, let w be the rightmost assigned node among the descendants of the nodes in $\{u_0, u_1, \ldots, u_i\}$. Let R be the set of nodes in $\{u_0, u_1, \ldots, u_i\}$ that are to the right of the level-l ancestor $\mathtt{anc}_l(w)$ of w. Then,

$$\mathtt{fc}\big(\{u_0, u_1, \ldots, u_i\}\big) = \mathtt{fc}\big(\mathit{Left}\big(\mathtt{anc}_l(w)\big)\big) + \mathtt{fc}\big(\{\mathtt{anc}_l(w)\}\big) + \mathtt{fc}(R). \tag{1}$$

By the choice of w, every node in R does not have any assigned descendant and thus is an assigned node or has an assigned ancestor; it follows that $\mathtt{fc}(R) = 0$. Note that $\mathtt{fc}(\mathit{Left}(\mathtt{anc}_l(w))) \leqslant \mathtt{fc}(\mathit{Left}(w))$ and since A is dense and w is assigned, there is no free node in $\mathit{Left}(w)$; by the induction hypothesis, we conclude that $\mathtt{fc}(\mathit{Left}(w)) < 2^{\mathtt{lv}(w)}$. Since $\mathtt{anc}_l(w)$ has at least one level $\mathtt{lv}(w)$ assigned descendent (namely w), we have $\mathtt{fc}(\{\mathtt{anc}_l(w)\}) \leqslant 2^l - 2^{\mathtt{lv}(w)}$. Substituting these bounds in (1), the lemma follows. □

Lemma 1 asserts that in a dense assignment, if we cannot find any free node at level l, then $\mathtt{fc}(\{u_0, u_1, \ldots, u_m\}) = \mathtt{fc}(\{\sigma\}) < 2^l$, and the total demand of the node-assignment requests served by A is greater than $2^h - 2^l$; together with Fact 1, we conclude that it is not possible to serve any level-l request. This is somewhat surprising because all previous algorithms need to maintain a more demanding global property (such as sorted and compact) to ensure full utilization of the tree capacity.

There is a simple algorithm to maintain dense assignments: to serve a level-l node-assignment request, we simply assign the leftmost free level-l node. However, this simple algorithm may not be competitive. The main difficulty comes from node releases. We identify two specific problems in Fig. 2(a) and (b). The sorted and compact configuration has the problem shown in the configuration in Fig. 2(b), which requires $O(h)$ swaps in the worst case for maintaining the dense property when node u is released. If the nodes are sorted in the other direction, i.e., with the levels sorted in non-increasing order, the case given in Fig. 2(a) arises when node u is released, where $O(n)$ swaps might be needed. Below, we define a configuration where the nodes are not sorted so that only a constant number of swaps are required.

Please cite this article in press as: Y. Zhang et al., Constant-competitive tree node assignment, Theoretical Computer Science (2013), http://dx.doi.org/10.1016/j.tcs.2013.05.014

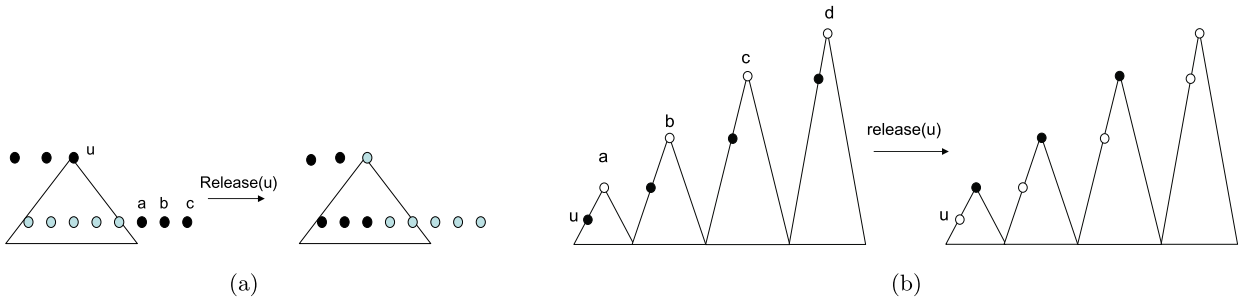


Fig. 2. Many swaps for maintaining denseness after a release.

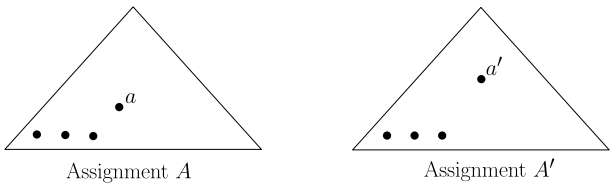


Fig. 3. No more than one safe assignment.

We first give some definitions to identify the two problematic configurations given in Fig. 2.

- For any two assigned nodes u and v, we say that v is a *tail* of u if (i) $\texttt{lv}(u) > \texttt{lv}(v)$ and (ii) u is to the left of v. We call the leftmost tail of u its first tail, the following tail to its right (if exists) its second tail, and so on. In Fig. 2(a), u has three tails a, b and c.
- For any node x at level l, we say that $T(x)$ is a *meager l-tree* if x has exactly one assigned descendant (and this implies that x must not be assigned). Fig. 2(b) shows four meager trees, namely $T(a), T(b), T(c)$, and $T(d)$.

We say that a node u is *safe* if (i) u has at most one tail, and (ii) there is no meager $\texttt{lv}(u)$-tree to its left. We say that an assignment A is *safe* if A is dense and all its assigned nodes are safe.

Fact 2. In a safe assignment, there is at most one meager l-tree for every level l.

In a safe assignment, any assigned node u has at most one tail and there is no meager $\texttt{lv}(u)$-tree to u's left. For any node a, we say that

- a is *dangerous to its left* if the following is satisfied: a is an assigned node and it is the second tail of some node x. We also say that a is dangerous to x, otherwise, we say a is *safe* to its left;
- a is *dangerous to its right* if the following is satisfied: a is an assigned node and it has an ancestor x such that (i) $T(x)$ is a meager $\texttt{lv}(x)$-tree, and (ii) there is an assigned level-$\texttt{lv}(x)$ node y to the right of x. We also say that a is dangerous to y, otherwise, a is *safe* to its right.

By considering the definition of *safe* assignment, we have the following fact:

Fact 3. An assigned node u is *safe* if and only if any other assigned node is *safe* (not *dangerous*) to u.

Now we show that there is at most one safe assignment for any fixed set of node-assignment requests.

Theorem 1. *Given a set of node-assignment requests, say S, there is at most one safe assignment that satisfies all requests in S.*

Proof. We prove this theorem by contradiction. Suppose there are two different safe assignments A and A' for satisfying all requests in S. From left to right, let a and a' be the first different assigned nodes in A and A' respectively. That means all assigned nodes to the left of a in A are same with those assigned nodes to the left of a' in A'. Let r_a and $r_{a'}$ be the requests for assigned nodes a and a' respectively. Without loss of general, assume that $\texttt{lv}(a) < \texttt{lv}(a')$, as shown in Fig. 3.

The assigned node x for request r_a must be to the right of a' in A', thus, x is a tail node of a'. Since a' has at most one tail node, the level of any assigned node to the right of a' except x must be no lower than $\texttt{lv}(a')$. In assignment A, the level of any assigned node to the right of a is strictly higher than $\texttt{lv}(a)$. Since both A and A' are dense, there is no free

Please cite this article in press as: Y. Zhang et al., Constant-competitive tree node assignment, Theoretical Computer Science (2013), http://dx.doi.org/10.1016/j.tcs.2013.05.014

level-$\mathtt{lv}(a)$ node to the left of a in A and to the left of a' in A'. Thus, a is the only assigned node in a meager $\mathtt{lv}(a')$-tree, which is to the left of assigned node for request $r_{a'}$ in A, contradicting the assumption that both A and A' are safe. □

By Fact 1, if the total demand of a set of node-assignments is more than 2^h, there is no legal assignment. In Theorem 2, we show that if the total demand of a set of node-assignments is no more than 2^h, our algorithm can satisfy all requests and the safe assignment can be maintained in satisfying each request. Therefore, if the total demand of a set of node-assignments is no more than 2^h, the safe assignment is unique.

3. An 8-competitive algorithm

Since there is at most one safe assignment for a set of node-assignment requests, our target is to find the operations that maintain the safe assignment for each request and the cost is minimized. In this section, we describe a competitive algorithm for maintaining safe assignment. It has two procedures *Node-Assignment* and *Node-Release*. We first describe Node-Assignment. It makes use of the following subroutine: *Packing*(v, r), where r is a level-l request and v is a free level-l node.

Procedure Packing(v, r)

1: **if** A meager l-tree $T(x)$ exists to the left of v **then**
2: Assign the root x of $T(x)$ to r and swap the single assigned node w in $T(x)$ with the leftmost free node at $\mathtt{lv}(w)$, which will become a tail node.
{By Fact 2, there is exactly one meager l-tree $T(x)$ in this case.}
3: **else**
4: Assign v to r.
5: **end if**

The following are the details of Node-Assignment.

Procedure Node-Assignment for serving a level-l request r
Case 1: [no free node at level l] Report "not enough bandwidth"
Case 2: [leftmost free level-l node a exists]
Case 2a: [The assigned node b immediately to the left of a is a tail of some node]
If assigning a will result in a second tail of some node
Let c be the leftmost assigned node with level $> \{\mathtt{lv}(b), \mathtt{lv}(a)\}$ [b is a tail of c]
(such node c must exist since there is at least one node which is to the left of b and with level higher than $\mathtt{lv}(b)$ and $\mathtt{lv}(a)$)
Swap c with the level-$\mathtt{lv}(c)$ common ancestor of a and b
EndIf
Release b from the request r_b to which it is currently assigned
Case 2a(i): [$\mathtt{lv}(a) \geqslant \mathtt{lv}(b)$]
Assign the leftmost free node at level $\mathtt{lv}(a)$ to r
Packing(the leftmost free node at level $\mathtt{lv}(b), r_b$)
Case 2a(ii): [$\mathtt{lv}(a) < \mathtt{lv}(b)$]
Assign the leftmost free node at level $\mathtt{lv}(b)$ to r_b
Packing(the leftmost free node at level $\mathtt{lv}(a), r$)
Case 2b: [either a is the leftmost node at level $\mathtt{lv}(a)$, or the assigned node b immediately to the left of a is not a tail of any node, and a is safe to its left by Lemma 2]
Packing(a, r)
end Node-Assignment

We now prove that if assignment A is safe, then after calling Node-Assignment to serve a request, the resultant assignment $\hat{A}$ is also safe.

Lemma 2. *Suppose that assignment A is safe. Let a be the leftmost free node at level $\mathtt{lv}(a)$. If the assigned node (not necessarily at level $\mathtt{lv}(a)$) immediately to the left of a is not a tail, then a is not dangerous to its left.*

Proof. Suppose to the contrary that a is dangerous to some node x to its left. Then, $\mathtt{lv}(x) > \mathtt{lv}(a)$, and x has a tail a' between x and a. We claim that the tree $X = T(\mathtt{anc}_{\mathtt{lv}(x)}(a'))$ has exactly one assigned node; otherwise x has more than one tail and A is not safe.

Please cite this article in press as: Y. Zhang et al., Constant-competitive tree node assignment, Theoretical Computer Science (2013), http://dx.doi.org/10.1016/j.tcs.2013.05.014

(a) A: Before serving the request (b) $\hat{A}$: After serving the request

Fig. 4. Case 2a(i) when Packing has no swapping.

Since A is safe, a' must be the leftmost level-$\mathtt{lv}(a')$ node in the tree X. Thus, X has at least one free node at $\mathtt{lv}(a)$, which leads to the contradiction to the facts: (1) a is the leftmost free node at level $\mathtt{lv}(a)$, and (2) the assigned node immediately to the left of a is not a tail. □

Lemma 3. *Suppose that the current assignment A is safe. After calling Node-Assignment to serve a level-l request r, the resultant assignment $\hat{A}$ is also safe.*

Proof. In Case 1, since $\hat{A} = A$, the resultant assignment $\hat{A}$ is safe.

To show the correctness of this lemma, all other cases should be verified. However, these cases can be analyzed by similar method, this is because

- Some cases are quite similar.
  The difference of Case 2a(i) and Case 2a(ii) is the order of assignment of node a and b;
- Some cases can be regarded as a special case of other.
  In Case 2b, the assigned node b immediately to the left of a is not a tail of any node, the *Packing*(a, r) in this case can be regarded as the last *Packing* step in Case 2a(ii).

Therefore, we will only consider Case 2a(i) in the following. In this case, *Packing*(the leftmost free node at level $\mathtt{lv}(b), r_b$) has not done any swapping, otherwise, there must exist an assigned node, which is dangerous to b in assignment A. In this case, the assigned node b immediately to a's left is a tail of some node c. Fig. 4(a) shows the positions of these nodes. If the assignment of node a will not result a second tail of some assigned node c, node c will be ignored in the procedure. Otherwise, b is the first tail while a is the second tail of c. Since a is the leftmost free level-ℓ node, a and b must lie in the same subtree rooted on c' such that $\mathtt{lv}(c') = \mathtt{lv}(c)$, as shown in Fig. 4(a).

According to Procedure Node-Assignment, and using the fact that A is dense, the new assignment $\hat{A}$ is shown in Fig. 4b. We say that $\hat{A}$ is dense because of the following:

- Since c and c' are not free in both A and $\hat{A}$, and we have not changed other nodes at $\mathtt{lv}(c)$. Together with the fact that $\mathtt{lv}(c)$ is dense in A, we conclude that $\mathtt{lv}(c)$ is dense in $\hat{A}$.
- Since c is safe in A, it has at most one tail and thus there is no assigned level-$\mathtt{lv}(b)$ node to the right of b or between c and c'. Hence, b' is the last assigned level-$\mathtt{lv}(b)$ node in $\hat{A}$. Since b is assigned in A and A is dense, there is no free node at level $\mathtt{lv}(b)$ preceding b in A and thus, there is no free node preceding b' in $\hat{A}$ (as b' is to the left of b). It follows that level-$\mathtt{lv}(b)$ is dense in $\hat{A}$.
- Since a is the leftmost free level-$\mathtt{lv}(a)$ node in A, and a' is to the left of a, we conclude that there is no free node at level $\mathtt{lv}(a)$ preceding a' in $\hat{A}$. Together with the fact that a' is the last assigned node at level-$\mathtt{lv}(a)$ (because c is safe in A), we have that $\mathtt{lv}(a)$ is dense in $\hat{A}$.
- For other levels, we claim that no free node appears to the left of an assigned node. From the procedure, only when some higher level node, say d, swapped to the right, a free node at lower level, say l', will appear. If there is an assigned level-l' node x to the right, this node must be a tail node of d. According to the algorithm, there is only one case for higher level assigned node swapped to the right, i.e., $d = c$, which has been proved to be dense. Therefore, all other levels are dense in $\hat{A}$.

We now verify that every assigned node in $\hat{A}$ is safe. Let O be the set of assigned nodes in $\hat{A}$ that are not a', b' and c'. The assigned nodes in O are also assigned nodes in A.

- We first prove that in $\hat{A}$, the newly assigned nodes a', b' and c' are not dangerous to any node in O.
  According to the algorithm, we have the following two observations:
  (i) In $\hat{A}$, every node in O that is to the right of c has level no smaller than $\mathtt{lv}(c)$; otherwise c is not safe in A.
  (ii) By the choice of c, there is no assigned node to the left of c (in both A and $\hat{A}$) with level greater than $\mathtt{lv}(a) = \mathtt{lv}(a') \geqslant \mathtt{lv}(b) = \mathtt{lv}(b')$.
  For any level $l > \mathtt{lv}(a')$, $X = T(\mathtt{anc}_l(a'))$ has at least two assigned descendants in $\hat{A}$, namely a' and b'; hence X is not a meager tree. Together with (i), we conclude that a' and b' are not dangerous to their right. By (ii), we conclude that

a' is not dangerous to their left. For b', since b is not a second tail in A, b' is not a second tail in A' too, thus, b' is not dangerous to the left. For c', if c' is dangerous to its right (left) in $\hat{A}$, then b would be dangerous to its right (left) in A. Since A is safe, c' is not dangerous to its left nor right.

- We then prove that in $\hat{A}$, any node in O is not dangerous to other node in O.
  If there is a node x which is dangerous to its right node y such that $x, y \in O$, then in $\hat{A}$, the subtree rooted at $\mathtt{anc}_{\mathtt{lv}(y)}(x)$ is a meager $\mathtt{lv}(y)$-tree to the left of y. Since the positions of x and y in $\hat{A}$ are same to the positions in A, recall that A is safe, there must exist another assigned node in the subtree rooted at $\mathtt{anc}_{\mathtt{lv}(y)}(x)$ in A. The difference of A and $\hat{A}$ is on the nodes a, b and c, however, the movements of these nodes do not satisfy the above condition. Contradiction! Thus, in $\hat{A}$, no node in O is dangerous to its right w.r.t. any other node in O.
  If there is a node x which is dangerous to its left node y such that $x, y \in O$, then in $\hat{A}$, x is the second tail of y. From A to $\hat{A}$, the positions of x and y do not change. Since A is safe, we have the following observations on the assignment A: (1) x is the only tail node of y, (2) x is located at the leftmost level-$\mathtt{lv}(x)$ node in the subtree rooted at $\mathtt{anc}_{\mathtt{lv}(y)}(x)$, and (3) the level of any assigned node in between y and x is no less than $\mathtt{lv}(y)$. Thus, from A to $\hat{A}$, there is an assigned node between y and x which is swapped out and a lower level node swapped in. Again, the difference of A and $\hat{A}$ is on the nodes a, b and c, and the movements of these nodes do not satisfy the above condition. Contradiction! Thus, in $\hat{A}$, no node in O is dangerous to its left w.r.t. any other node in O.

According to Fact 3, any node in O is safe in the assignment $\hat{A}$.

We now show that the remaining three assigned nodes a', b' and c' are also safe. Note that a', b', c' do not have more than one tail node because of (i). There is no meager $\mathtt{lv}(a')$-tree to the left of a' in $\hat{A}$, otherwise, if $\mathtt{lv}(a') > \mathtt{lv}(b')$, A is not safe since level $\mathtt{lv}(b)$ is not dense; and if $\mathtt{lv}(a') = \mathtt{lv}(b')$, A is not safe since there is a meager $\mathtt{lv}(b)$-tree to the left of b in A. There is no meager $\mathtt{lv}(b')$-tree to the left of b' in $\hat{A}$ because there is no such tree to the left of b in A. There is no meager $\mathtt{lv}(c')$-tree to the left of c' in $\hat{A}$ because of (1) in $\hat{A}$, every node in O that is to the right of c has level no smaller than $\mathtt{lv}(c)$; (2) there is no meager $\mathtt{lv}(c)$-tree to the left of c in A; and (3) a and b are in the same subtree rooted at level $l > \mathtt{lv}(a')$.

Combining the above analysis, $\hat{A}$ is dense and all its assigned nodes are safe, thus, this lemma is true. □

The release operation, listed in the following, can be regarded as the reverse of the assignment operation.

Procedure Node-Release for a level i node a
If a is not the rightmost assigned node at level i
  Swap the rightmost assigned node to a at level i (**1 cost**)
End If
If there exists a tail node c to the right of a and with level $i' < i$
  Swap c to the leftmost free node at level $\mathtt{lv}(c)$ (**1 cost**)
(The reverse of Packing in Case 2 of Procedure Node-Assignment)
End If
If there exists a meager l-tree to the left of level-l assigned node
(There is at most one such meager l-tree by the swap of tail node c. From the structure of safe assignment, $l > i$)
  Swap the meager l-tree and the rightmost assigned node at level l (**2 costs**)
(The reverse of Case 2a of Procedure Node-Assignment without Packing)
End If

Lemma 4. *Suppose that the current assignment $\hat{A}$ is safe. After calling Node-Release for a level-l node a, the resultant assignment A is also safe.*

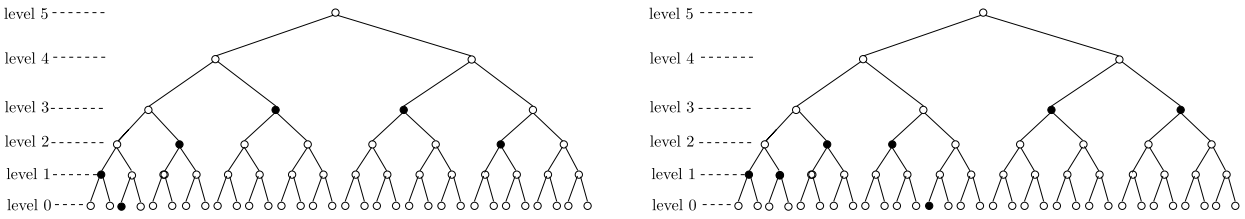
Proof. From Theorem 1, the safe assignment for a set of node-assignment requests is unique. A safe assignment A and an assignment request r uniquely determine a resultant safe assignment $\hat{A}$. We can also say that a safe assignment A is deterministically decided by a safe assignment $\hat{A}$ and a release request r. □

Theorem 2. *Node-Assignment and Node-Release can correctly solve the tree node assignment problem.*

Proof. Since Node-Release is the reverse of Node-Assignment, we may only analyze the algorithm Node-Assignment.

We claim that Node-Assignment can satisfy any assignment request if the total capacity is not larger than the capacity of the whole tree, i.e., the summation of the assigned capacity C and the requested capacity c is no more than 2^h, where h is the height of the tree. If Node-Assignment cannot satisfy a level i request r in a safe assignment A, since A is dense and no free node at level i, from Lemma 1, the total free capacity $\mathtt{fc}(\sigma) < 2^i$.

Please cite this article in press as: Y. Zhang et al., Constant-competitive tree node assignment, Theoretical Computer Science (2013), http://dx.doi.org/10.1016/j.tcs.2013.05.014



(a) before node assignment at level 1 (b) before node release at level 1

Fig. 5. Worst-case example for competitive ratio of 8.

Now we show that any situation is considered in Node-Assignment. If the total free capacity is large enough for the assignment request, From Lemma 2, if Case 2a of Node-Assignment does not hold, then the assertion in Case 2b must be executed. □

Theorem 3. *Our algorithm is* 8*-competitive for the tree node assignment problem and this bound is tight for our algorithm.*

Proof. From the Procedures Node-Assignment and Node-Release, it can be verified that each request and release operation uses at most four assignments/swaps, each *Packing* operation takes at most one assignment and one swap. For a given sequence of node assignments/releases comprising a mix of m_1 node assignments and m_2 node releases, the total cost of the algorithm is at most $4m_1 + 4m_2$. Since $m_1 \geqslant m_2$ and m_1 gives a lower bound on total cost, the competitive ratio of our algorithm is at most 8.

In the following, we give a node request sequence on a *safe* configuration to show that the competitive ratio of 8 is the best bound for our algorithm. Specifically, in this example, we will show that each node assignment/release costs 4 assignments/swaps while the optimal cost for each node assignment is 1 and for each node release is 0.

There are two *safe* configurations shown in Fig. 5, which are related in the following way: a node assignment at level 1 would make the *safe* configuration in Fig. 5(a) change to the one in Fig. 5(b); and alternatively, a node release at level 1 would make the *safe* configuration in Fig. 5(b) change back to Fig. 5(a). Consider a sequence σ of node assignment/release requests on the configuration shown in Fig. 5(a). $\sigma = \{c_1, c_2, \ldots, c_k, \ldots\}$ where c_{2i-1} is a node assignment at level 1 and c_{2i} is a node release for earlier assigned node at level 1. According to the algorithm, each node assignment/release would require 4 assignments/swaps, while the cost of each node assignment and release of the optimal offline algorithm is one and zero, respectively. Thus, we can conclude that 8 is a tight bound for the competitive ratio of the algorithm. □

4. An improved 6-competitive algorithm

This section introduces a *lazy-release* technique, which allows us to improve our 8-competitive algorithm to a 6-competitive algorithm. The major change is the following: When an assigned node is released, the released node is marked as a *full hole* and no node is swapped to keep the assignment *safe*; the *full holes* will be assigned by the procedure *Compact* when needed. The handling of node assignment is similar to Node-Assignment in previous section, thus it takes 4 costs per request. By implementing *lazy-release*, the cost for each node release request is 0. The cost of *Compact* is not a constant. However, it is surprising that the cost of *Compact* can be covered by associating two credits to each node release request. Therefore, a 6-competitive algorithm can be obtained.

When serving a node assignment request, or implementing the procedure *Compact*, we may need to swap an assigned node v with a free node (full hole). In such case, the newly freed node v will be marked as a *half hole*. If a free node u has two descendants which are labeled with *half hole* or *full hole*, in the processing of the procedure *Compact*, these two holes at lower levels will disappear and node u will be marked as *full hole*. In the later part, we use *hole* to present both *full hole* and *half hole* in the case of no confusion.

We say that a legal assignment is *virtually safe* if (i) after treating all full holes and half holes as assigned nodes, the resultant assignment is safe, and (ii) each half hole is to the right of all assigned nodes at the same level. Obviously, a safe assignment is also virtually safe.

For any node u, let T_{uL} be the portion of T that comprises of the subtree rooted at u, all u's ancestors and their left subtrees that do not contain u. Let T_{uR} be the part of T formed by the nodes not in T_{uL}. Fig. 1 gives an example of T_{dL} and T_{dR}. We extend our definition of $\mathtt{fc}$ such that $\mathtt{fc}(T_{uL})$ is the set of free leaves in T_{uL}. For any level i, we define

$$free_i = \max_u \left\{ \mathtt{fc}(T_{uL}) \;\middle|\; T_{uL} \text{ does not contain any assigned node or hole whose level is higher than } i \right\}$$

where u can be any node in the tree. For convenience, we may let the range of u to be all leaves.

With the help of $free_i$, when we do the *compact*(i) in New-Node-Assignment, the operations only affect the assigned nodes from level $0 < k < i$ to i such that the resultant configuration from level k to level i is sorted-and-compact except at most one tail node in level k. Moreover, since we have no need to swap assigned nodes and consider tail nodes in levels lower than k, the costs will be saved. For the assignment shown in Fig. 6, we have $free_0 = 2$, $free_1 = free_2 = 7$, $free_3 = \mathtt{fc}(\sigma) = 11$.

Please cite this article in press as: Y. Zhang et al., Constant-competitive tree node assignment, Theoretical Computer Science (2013), http://dx.doi.org/10.1016/j.tcs.2013.05.014

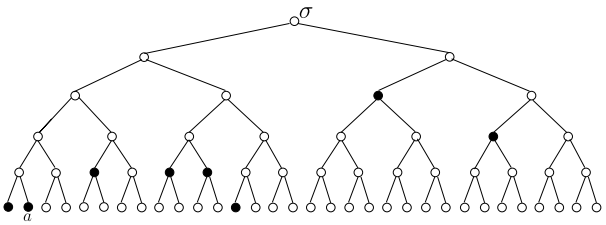


Fig. 6. An example for $free_i$.

Now we give the modified algorithm by implementing *lazy-release*. As described before, when a node is released, we do nothing but just label this node as a "full hole". When a node assignment request on level i comes, we first try to satisfy this request by a hole in level i. If there is no level-i hole in the assignment, we regard each hole as assigned node, then apply Node-Assignment. If the above two steps cannot satisfy the request, we will try to use the hole at level higher than i. If there is no hole at higher level and the total free capacity is large enough to satisfy the request, i.e., $free_i \geqslant 2^i$, we do the procedure *Compact* to free up a node at level i. The algorithm is formally described as follows.

Procedure New-Node-Assignment
for serving a level-i request r in a virtually safe assignment A

1. [level-i hole exists in A:] assign the leftmost hole to r and return.
2. [no level-i hole in A:] Regard all holes in A as assigned nodes, then perform Node-Assignment to satisfy r. If Node-Assignment successfully assigns a node v to r, return.
3. [no level-i free node in A, but level-j hole, $j > i$, exists in A:] Let w be the leftmost hole at level $j > i$. Let x be the leftmost assigned node at level j. If x is to the left of w, swap x and w, which makes x a free node. Let z be the current leftmost free node at level j, i.e., either x or w. Mark z's left child y as a full hole, and perform *Packing*(*the leftmost free node at level i to the right of y*, r). Then return.
4. [no level-i free node in A, no level-j hole, $j > i$, in A:] If $free_i \geqslant 2^i$, perform *Compact*(i) and assign the leftmost free node at level i to r. Then return.
5. [Otherwise, no free node at level i and $free_i < 2^i$] Report "not enough bandwidth".

end New-Node-Assignment

We now give details of *Compact*(i). It will invoke the procedure *Fillfree*(l), which simply repeatedly swaps the rightmost assigned node to the leftmost free node (full hole), which must be to the left of the rightmost assigned node, at level l until the assigned nodes at level l are dense. *Compact*(i) is supposed to compact the assigned nodes at levels lower than i so as to free up a level-i node. On the other hand, we would like to get rid of as many tail nodes as possible. Note that *Compact*(i) is called when $free_i \geqslant 2^i$.

Procedure *Compact*(i)

1: Find the highest tail node c (of some assigned node or hole) at level $k \leqslant i$ such that $free_k < 2^k$, if no such c exists, set $k = -1$.
   {We claim that $k < i$; otherwise, there is a tail node at level i, which means level i contains a free node and the procedure would be terminated in Step 2.}
2: Release c (if exists)
3: **for** $l = k + 1$ to $i - 1$ **do**
4: *Fillfree*(l)
5: **end for**
6: **if** $k \geqslant 0$ and the tail node c exists **then**
7: **if** the leftmost free node at $\mathtt{lv}(c)$ is in an empty subtree rooted at a level i node u **then**
8: Assign c to the leftmost free level-$\mathtt{lv}(c)$ node in the subtree rooted at the right-son of u. The left-son of u will be regarded as a hole.
9: **else**
10: Assign c to the leftmost free node at $\mathtt{lv}(c)$.
11: **end if**
12: **end if**

end Compact

To have a better understanding of the procedure *Compact*, we may consider the example as shown in Fig. 7. Our target is to serve a level-4 request, however, there is no free node at level 4. Since $free_4 > 2^4$, the procedure Compact will be invoked to free up a level 4 node. The left part is the assignment before *Compact*, level 0 and level 2 both contain a tail node, we

Please cite this article in press as: Y. Zhang et al., Constant-competitive tree node assignment, Theoretical Computer Science (2013), http://dx.doi.org/10.1016/j.tcs.2013.05.014

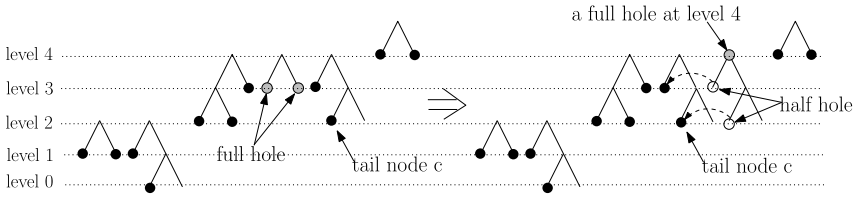


Fig. 7. An example of the procedure *Compact*.

set $k = 2$. There are two full holes at level 3. After Compact, the assignment is changed as shown in the right part. There are two node swaps at level 3 and level 4, thus, two half holes appear, and the common ancestor at level 4 of these two half holes will be regarded as a full hole.

Consider any consecutive levels $i, i+1, \ldots, j$. We say that these levels are *sorted and compact* if for each level $l \in [i, j]$, the assigned nodes at level l are consecutive and to the left of any assigned node at higher levels, say level no lower than $l+1$, and there is no free level-l node which is to the left of some assigned level-l node. The following lemma shows some properties about the configuration after Procedure New-Node-Assignment has called Procedure *Compact*(i).

Lemma 5. *Suppose the configuration before Compact*(i) *is virtually safe, then after Compact*(i) *is called, the assigned nodes at levels from* $k+1$ *to* $i-1$ *are sorted and compact, and there is at most one tail node at level* k *and at least one free node at level* i.

Proof. In performing *Compact*(i), the tail c at level k (if exists) is released first, then we call *Fillfree* from level $k+1$ to level $i-1$. In the case where k is set to -1, the procedure *Fillfree* starts from level 0.

Firstly, consider level $k+1$, we will prove that after *Fillfree*$(k+1)$, level $k+1$ is compact, i.e., the assigned nodes at level $k+1$ are consecutive.

- If there was no tail node at level $k+1$, then all level-$(k+1)$ assigned nodes will finally be swapped to be consecutive. Otherwise, suppose there is a level-j assigned node between two level-$(k+1)$ assigned nodes, then j must be smaller than $k+1$. This level-j node must exist between two level-$(k+1)$ assigned nodes or holes in the previous assignment, but this will lead to the conclusion that it would be dangerous to either its left if there was more than one such level-j node or its right if there was only one such level-j node.
- If there was a tail node x at level $k+1$ (according to A), then there must be a hole at level $k+1$ in A. Otherwise, from the definition of $free_i$, we have $free_{k+1} < 2^{k+1}$ since $free_k < 2^k$, contradicting the selection of level k. Thus, we can always move the tail node to join the preceding nodes. Then, similar to the above argument in case of no tail node, we can claim that in the final assignment, the assigned nodes at level $k+1$ are consecutive.

We can repeat exactly the same argument to show that after *Fillfree*$(k+2)$, *Fillfree*$(k+3), \ldots,$ *Fillfree*$(i-1)$, all these levels become compact, and since we move nodes at lower levels first, they are also sorted. The tail node c must be immediately to the right of an assigned node (or a hole) at level in between $k+1$ and i, otherwise, there exists a free node at level i and *New-Node-Assignment* will be terminated at Step 2. After *Compact*(i), c must be assigned immediately to the right of some assigned node at level in between $k+1$ and i. Now, we apply the argument to i, and then we conclude that after *Compact*(i), there must exist a free node at level i. Otherwise, since $free_k < 2^k$ and level $k+1$ to level $i-1$ are sorted and compact after *Compact*(i), the total free capacity $\mathtt{fc}(T_{uL})$ such that $\mathtt{fc}(T_{uL}) = free_i$ is no more than $\sum_{j=k}^{i-1} 2^j < 2^i$. Note that *Compact*$(i)$ does not affect the value of $free_i$, thus, no free node at level i after *Compact*(i) contradicts with $free_i > 2^i$. □

Theorem 4. *After calling New-Node-Assignment to serve a request in a virtually safe assignment, the resultant configuration is virtually safe.*

Proof. If we do the assignment by Step 1 or 2 in New-Node-Assignment, condition (i) obviously holds. In Step 1, condition (ii) also holds. In Step 2, we claim that condition (ii) holds, too, since we never swap a hole with a left assigned node at the same level.

If the update is from Step 3, it is straightforward to verify that after the update, in the resultant assignment, every node is virtually safe, i.e., by assuming all the holes as assigned nodes, they are safe. It is easy to check that condition (ii) holds after the update.

For Step 4, after *Compact*(i), all the levels from $k+1$ to $i-1$ are dense. Furthermore, all the assigned nodes from $k+1$ to $i-1$ are sorted and compact (Lemma 5). The assigned nodes/holes at levels lower than k and higher than i are kept intact. For every level $k+1 \leqslant \ell \leqslant i-1$, there is at most one free node, whose sibling is assigned or not free by the assigned node at lower level. We now claim that this kind of free node is not dangerous to the right. Otherwise, before *Compact*(i), the subtree containing this node was also dangerous to the right, since *Compact*(i) only moves the assigned nodes to the left and keeps half holes to the right (half holes may be moved to upper levels and changed into full holes without violating

Please cite this article in press as: Y. Zhang et al., Constant-competitive tree node assignment, Theoretical Computer Science (2013), http://dx.doi.org/10.1016/j.tcs.2013.05.014

virtually-safe property). Before assigning the tail node c, every level in the assignment is virtually safe. Since c is assigned between the assigned nodes at levels $k+1 \leqslant \ell \leqslant i$, after assigning c, every level in the assignment is also virtually safe. Then assign r the leftmost free node at level i. This leads to a virtually safe assignment, since the validity condition (ii) can be verified. □

In performing *Compact*, the number of swaps in each *Fillfree* operation is not fixed. We may have to associate some credits with each hole upon any node release to cover these swaps. Thus, the cost of each swap of an assigned node to a free node on the left (which must be a full hole, otherwise, the previous assignment is not virtually safe) can be covered by the associated credits of the holes. Intuitively, suppose there are k holes at level i. After k swaps, the assigned nodes at level i are compact, but at most $k/2$ new free nodes (holes) may appear at level $i+1$. If we associate 2 credits with the full hole and there is a swap later, these 2 credits could cover the cost of the swap plus half of the two credits (i.e. one credit) required by the free node at higher level half-created by this swap. After *Compact*, there is at most one full/half hole at each affected level, whose sibling is not free.

Now we give the correctness proof of New-Node-Assignment, i.e., the bandwidth is fully utilized by this algorithm.

Lemma 6 *(Correctness). Given a virtually safe assignment, a level-*i *node request can be satisfied by New-Node-Assignment if and only if the total free capacity* (*i.e. the total capacity of all free leaf nodes*) *is not less than* 2^i.

Proof. [Only If] If the strategy can satisfy the level-i assignment, obviously, the total free capacity is not less than 2^i.

[If] Suppose the strategy cannot satisfy a level i assignment, i.e., there is no free level-i node (Steps 1, 2, and 3 cannot be invoked) and $free_i < 2^i$ (Step 4 cannot be invoked). Since no level-i free node exists, there is no free node with level $> i$. Consider any assigned node u with level higher than i, we claim that u has no tail node, otherwise, a free node at level i exists. Combine the above two statements and the virtually safe property, we have $free_i = free_{i+1} = \cdots = \mathtt{fc}(\sigma)$. From the assumption that $free_i$ is strictly less than 2^i, the total free capacity in the whole node tree is strictly less than 2^i. □

In a standard way of amortized analysis, we charge for every assignment/swap 1 credit. In the following lemma, we show that by associating 4 and 2 credits to the assignment request and release request respectively, the number of swaps in the procedure Compact can be covered by these credits.

Lemma 7. *If each node assignment and release is associated with* 4 *and* 2 *credits respectively, then the full holes and half holes will have at least* 2 *and* 1 *credits respectively, and the number of swaps of Compact can be covered by the credits associated with the holes plus at most three more credits.*

Proof. We prove this lemma by induction on the level number i. The base step is *Compact*(1), since for level 0, if there is enough capacity ($f_\sigma \geqslant 2^0$), the level-0 request can be always satisfied in Step 1 or Step 2 in New-Node-Assignment. In the base step, there is no hole at level 1, each full hole at level 0 must be from a release and thus has 2 credits; and each half hole must be from a swap of an assigned node and a full hole and thus has 1 credit. *Compact*(1) might require at most 3 credits to handle the tail node – 2 for node release (credits associated with a hole) and 1 for node assignment. The swaps in the *Fillfree* operation at level 0 can be covered by the credits associated with the holes. Thus, this lemma is true for $i = 1$. Note that two swaps at level 0 may lead to two half holes, which may form a full hole with 2 credits at level 1.

Suppose this lemma is true for levels lower than i, i.e., each full hole, which comes from a node release or from the combination of two lower level holes, has 2 credits and each half hole, which is from a swap of a former full hole or from the movement of a lower level half hole, has 1 credit. Now we analyze the potential of holes at level i.

If a full hole at level i comes from a node release, it obviously has 2 credits. If a full hole at level i is formed from two lower level holes, from the induction hypothesis, this level i full hole can be regarded as having 2 credits. The proof that a half hole at level i has 1 credit is similar.

In *Compact*(i), the swap in the *Fillfree* operation can be covered by the credits associated with the holes. If there exists a tail node in the lowest affected level k, as in the base case, 3 more credits, two for a node release (credits associated with a hole) and one for the node assignment, are needed. □

Theorem 5. *Lazy-release is a* 6*-competitive algorithm for the tree node assignment problem.*

Proof. Now assume each node assignment and release are associated with 4 and 2 credits respectively. From the description of the first four steps of New-Node-Assignment, we claim that each step can be covered by 4 credits. Step 1 takes one credit and the Node-Assignment in Step 2 takes 4 credits. In Step 3, the total cost is at most 1 (swap of x) + 2 (credits of full hole on y) + 2 (*Packing*) − 1 (credit of hole on w) = 4. Since the *Compact* in Step 4 can be covered by the credits associated with holes plus at most three extra credits (Lemma 7), with one extra credit needed for the node assignment, 4 credits would be sufficient.

Similar to Theorem 3, for a given sequence of m_1 node assignments and m_2 node releases, the total cost of the algorithm is at most $4m_1 + 2m_2$. Since $m_1 \geqslant m_2$ and m_1 is a lower bound for the optimal number of node assignments and swaps, the competitive ratio of our lazy-release algorithm is 6. □

Please cite this article in press as: Y. Zhang et al., Constant-competitive tree node assignment, Theoretical Computer Science (2013), http://dx.doi.org/10.1016/j.tcs.2013.05.014

Acknowledgements

The authors thank Dr. Bethany M.Y. Chan for her efforts in making this paper more readable. We thank the editor and the anonymous referees for very helpful insights and comments.

References

[1] G.S. Brodal, E.D. Demaine, J.I. Munro, Fast allocation and deallocation with an improved buddy system, Acta Inform. 41 (4) (2005) 273–291.
[2] Joseph Wun-Tat Chan, Francis Y.L. Chin, Deshi Ye, Yong Zhang, Online frequency allocation in cellular networks, in: Proc. of the 19th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA 2007), pp. 241–249.
[3] Joseph Wun-Tat Chan, Francis Y.L. Chin, Hingfung Ting, Yong Zhang, Online tree node assignment with resource augmentation, J. Comb. Optim. 22 (3) (2011) 359–377. A preliminary version is in: Proceedings of the 15th International Computing and Combinatorics Conference (COCOON 2009), in: LNCS, vol. 5609, pp. 358–367.
[4] F.Y.L. Chin, H.F. Ting, Y. Zhang, A constant-competitive algorithm for online OVSF code assignment, Algorithmica 56 (1) (2010) 89–104.
[5] F.Y.L. Chin, Y. Zhang, H. Zhu, Online OVSF code assignment in cellular networks, in: Proc. of the third International Conference on Algorithmic Aspects in Information and Management (AAIM 07), in: LNCS, vol. 4508, pp. 191–200.
[6] D.C. Defoe, S.R. Cholleti, R.K. Cytron, Upper bound for defragmenting buddy heaps, in: Proc. of the 2005 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems, 2005, pp. 222–229.
[7] Shantanu Dutt, John P. Hayes, Subcube allocation in hypercube computers, IEEE Trans. Comput. 40 (3) (1991) 341–352.
[8] T. Erlebach, R. Jacob, M. Mihalak, M. Nunkesser, G. Szabo, P. Widmayer, An algorithmic view on OVSF code assignment, Algorithmica 47 (3) (2007) 269–298.
[9] T. Erlebach, R. Jacob, Marco Tomamichel, Algorithmische Aspekte von OVSF Code Assignment mit Schwerpunkt auf Offline Code Assignment, Student thesis at ETH Zürich.
[10] Michal Forisek, Branislav Katreniak, Jana Katreniakova, Rastislav Kralovic, Richard Kralovic, Vladimir Koutny, Dana Pardubska, Tomas Plachetka, Branislav Rovan, Online bandwidth allocation, in: Proc. of the 15th Annual European Symposium on Algorithms (ESA 2007), in: LNCS, vol. 4698, pp. 546–557.
[11] Xiang-Yang Li, Peng-Jun Wan, Theoretically good distributed CDMA/OVSF code assignment for wireless ad hoc networks, in: Proc. the 11th Annual International Conference of Computing and Combinatorics (COCOON 2005), pp. 126–135.
[12] Kenneth C. Knowlton, A fast storage allocator, Commun. ACM 8 (10) (1965) 623–624.
[13] Donald E. Knuth, The Art of Computer Programming, vol. 1: Fundamental Algorithms, Addison–Wesley, 1975.
[14] Shuichi Miyazaki, Kazuya Okamoto, Improving the competitive ratio of the online OVSF code assignment problem, in: Proc. of the 19th International Symposium on Algorithms and Computation (ISAAC 2008), pp. 64–76.

Please cite this article in press as: Y. Zhang et al., Constant-competitive tree node assignment, Theoretical Computer Science (2013), http://dx.doi.org/10.1016/j.tcs.2013.05.014