Algorithmica (2007) 47: 269–298
DOI: 10.1007/s00453-006-0188-3

Algorithmica

An Algorithmic View on OVSF Code Assignment[1]

Thomas Erlebach,[2] Riko Jacob,[3] Matúš Mihalák,[2]
Marc Nunkesser,[3] Gábor Szabó,[3] and Peter Widmayer[3]

Abstract. Orthogonal Variable Spreading Factor (OVSF) codes are used in UMTS to share the radio spectrum among several connections of possibly different bandwidth requirements. The combinatorial core of the OVSF code assignment problem is to assign some nodes of a complete binary tree of height h (the code tree) to n simultaneous connections, such that no two assigned nodes (codes) are on the same root-to-leaf path. A connection that uses a 2^{-d} fraction of the total bandwidth requires some code at depth d in the tree, but this code assignment is allowed to change over time. Requests for connections that would exceed the total available bandwidth are rejected. We consider the one-step code assignment problem: Given an assignment, move the minimum number of codes to serve a new request. Minn and Siu propose the so-called DCA algorithm to solve the problem optimally. In contrast, we show that DCA does not always return an optimal solution, and that the problem is *NP*-hard. We give an exact $n^{O(h)}$-time algorithm, and a polynomial-time greedy algorithm that achieves approximation ratio $\Theta(h)$. A more practically relevant version is the online code assignment problem, where future requests are not known in advance. Our objective is to minimize the overall number of code reassignments. We present a $\Theta(h)$-competitive online algorithm, and show that no deterministic online algorithm can achieve a competitive ratio better than 1.5. We show that the greedy strategy (minimizing the number of reassignments in every step) is not better than $\Omega(h)$ competitive. We give a 2-resource augmented online algorithm that achieves an amortized constant number of (re-)assignments. Finally, we show that the problem is fixed-parameter tractable.

Key Words. OVSF codes, Code assignment, Approximation algorithms, Online algorithms.

1. Introduction. Recently UMTS (Universal Mobile Telecommunications System, for more details see [17] and [20]) has received a lot of attention, and also raised new algorithmic problems. In this paper we focus on a specific aspect of its air interface W-CDMA (Wideband Code Division Multiple Access) that turns out to be algorithmically interesting, more precisely on its multiple access method DS-CDMA (Direct Sequence Code Division Multiple Access). The purpose of this access method is to enable all users in one cell to share the common resource, i.e. the bandwidth. In DS-CDMA this is accomplished by a spreading and scrambling operation. Here we are interested in the spreading operation that spreads the signal and separates the transmissions from the

[1] A preliminary version of this paper has appeared in the *Proceedings of the* 21*st International Symposium on Theoretical Aspects of Computer Science* (STACS '04), Lecture Notes in Computer Science 2996, Springer-Verlag, Berlin, March 2004, pp. 270–281. We gratefully acknowledge the partial support of ETH Zürich under project TH 46/02-1, "Mobile phone antenna optimization".
[2] Department of Computer Science, University of Leicester, University Road, Leicester LE1 7RH, England. {t.erlebach,mihalak}@mcs.le.ac.uk.
[3] Department of Computer Science, ETH Zürich, 8092 Zürich, Switzerland. {jacob,nunkesser,szabog, widmayer}@inf.ethz.ch.

Received December 17, 2003; revised September 19, 2005. Communicated by A. Panconesi.
Online publication March 13, 2007.

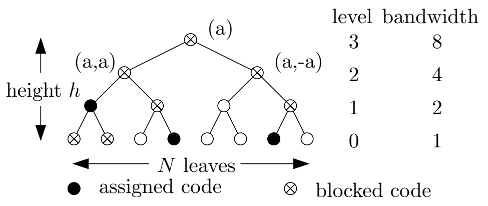


Fig. 1.1. A code assignment and blocked codes.

base-station to the different users. More precisely, we consider spreading by Orthogonal Variable Spreading Factor (OVSF) codes [2], [17], which are used on the downlink and the dedicated channel of the uplink. These codes are derived from a code tree. The OVSF-code tree is a complete binary tree of height h that is constructed in the following way: The root is labeled with the vector (1), the left child of a node labeled (a) is labeled with (a, a), and the right child with $(a, -a)$. Each user in one cell is assigned a different OVSF code. The key property that separates the signals sent to the users is the *mutual orthogonality* of the users' codes. All assigned codes are mutually orthogonal if and only if there is at most one assigned code on each leaf-to-root path. In DS-CDMA users request different data rates and get OVSF codes of different levels. (The data rate is inversely proportional to the length of the code. Note that a code at depth d has code length 2^d, since the code length of a child is twice the code length of the parent.) In particular, it is irrelevant which code on a level a user gets, as long as all assigned codes are mutually orthogonal. We say that an assigned code in any node in the tree *blocks* all codes in the subtree below it and all codes on the path to the root, see Figure 1.1 for an illustration. A maximal subtree of unblocked codes is called a *gap tree* (see Figure 7.6(a) in Section 7).

As users connect to and disconnect from a given base station, i.e. request and release codes, the code tree can get fragmented. Then it can happen that a code request for a higher level cannot be served at all, because lower level codes block *all* codes on this level. For example in Figure 1.1 no code can be inserted on level 2 without reassigning another code, even if there is enough available bandwidth. This problem is known as *code blocking* or *code-tree fragmentation* [20], [21]. One way of solving this problem is to reassign some codes in the tree (more precisely, to assign different OVSF codes of the same level to some users in the cell). In Figure 1.2 some user requests a code on level 2, where all codes are blocked. Still, after reassigning some of the assigned codes as indicated by the dashed arrows, the request can be served. Here and in many of the following figures we only depict the relevant parts (subtrees) of the single code tree.

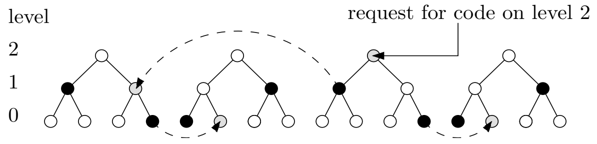


Fig. 1.2. A code insertion on level 2 into a single code tree T, shown without the top levels.

The process of reassigning codes necessarily induces signaling overhead from the base station to the users whose codes change. This overhead should be kept small. Therefore, a natural objective already stated in [21] and [23] is to serve all code requests as long as this is possible, while keeping the number of reassignments as small as possible. (In fact, as long as the total bandwidth of all simultaneously active code requests does not exceed the total bandwidth, it is always possible to serve them, see Corollary 2.2.) The problem has been studied before with the focus on simulations. In [21] the problem of reassigning the codes for a single additional request is introduced. The Dynamic Code Assignment (DCA) algorithm is presented and claimed to be optimal. In this paper we prove that this algorithm is not always optimal and analyze natural versions of the underlying code assignment (CA) problem. We present a first rigorous analysis of this problem.

After some preliminaries on the problem in Section 2 we give a counterexample to the optimality of the DCA algorithm in Section 3, then prove the original problem stated by Minn and Siu to be *NP*-complete for a natural, compact input encoding (where the initial code assignment is specified by a list of positions of assigned codes) in Section 4. In Section 5 we give a dynamic programming algorithm that solves the problem with running time $n^{O(h)}$, where n is the number of assigned codes in the tree. In Section 6 we give an involved analysis showing that a natural greedy algorithm already mentioned in [21] achieves approximation ratio h for the problem of assigning a single additional request. We tackle the online-problem in Section 7. It is a more natural version of the problem, because we are interested in minimizing the signaling overhead over a sequence of operations rather than for a single operation only. We present a $\Theta(h)$-competitive algorithm and show that the greedy strategy that minimizes the number of reassignments in every step is not better than $\Omega(h)$-competitive in the worst case. We also give an online algorithm with constant competitive ratio that uses resource augmentation, where we give our code tree one more level than the adversary. Finally, we show that the original problem is fixed-parameter tractable for some natural parameters, assuming a non-compact input encoding (where the initial code assignment is specified by a bit vector indicating for each position whether it has an assigned code or not).

1.1. *Problem Definition*. We consider the combinatorial problem of assigning codes to users. The codes are the nodes of an (OVSF) code tree $T = (V, E)$. Here T is a complete binary tree of height h. The set of all users using a code at a given moment in time can be modeled by a *request vector* $r = (r_0, \dots, r_h) \in \mathbb{N}^{h+1}$, where r_i is the number of users requesting a code on level i (with bandwidth 2^i). The levels of the tree are counted from the leaves to the root starting at level 0. We denote by $l(v)$ the level of node v. Each request is assigned to a position (node) in the tree, such that for all levels $i \in \{0, \dots, h\}$ there are exactly r_i codes on level i, and on every path p_j from a leaf j to the root there is at most one code assigned. We call every set of positions $F \subset V$ in the tree T that fulfills these properties a *code assignment*. For ease of presentation we denote by F the set of *codes*. Throughout this paper a code tree is the tree together with a code assignment F. If a user connects to the base station, the resulting additional request for a code represents a *code insertion* (on a given level). If some user disconnects, this represents a *deletion* (in a given position). A new request is dropped if it cannot be served. This is the case, if its acceptance would exceed the total bandwidth. By N we denote the number of leaves of T, i.e. $N = 2^h$, and by n the number of assigned codes $|F|$. After an insertion on level l_t at time t, any CA algorithm must change the code assignment F_t into F_{t+1} for the new request vector $r' = (r_0, \ldots, r_{l_t} + 1, \ldots, r_h)$. The size $|F_{t+1} \setminus F_t|$ corresponds to the number of *reassignments*. This implies that for an insertion, the new assignment is counted as a reassignment. We define the number of reassignments as the cost function. Deletions are not considered in the cost function, they are charged to the insertions. When we want to emphasize the combinatorial side of the problem we call a reassignment a *movement* of a code.

We state the original CA problem studied by Minn and Siu together with some of its natural variants:

One-step offline CA. Given a code assignment F for a request vector r and a code request for level l. Find a code assignment F' for the new request vector $r' = (r_0, \ldots, r_l + 1, \ldots, r_h)$ with a minimum number of reassignments.

General offline CA. Given a sequence S of code insertions and deletions. Find a sequence of code assignments so that the total number of reassignments is minimum, assuming the initial code tree is empty.

Online CA. In this setting requests are served as they arrive without knowledge of the future requests. The cost function is again the total number of reassignments over the whole request sequence.

Insertion-only online CA. This is the online CA with insertions only.

1.2. *Related Work.* It was a paper by Minn and Siu [21] that originally drew our attention to this problem. There the one-step offline version is defined together with an algorithm that is claimed to solve it optimally. As we show in Section 3 this claim is not correct, the argument contains errors. Many of the follow-up papers like [4], [6], [8], [10], [14], [15], [19] and [23] acknowledge the original problem to be solved by Minn and Siu and study some other aspects of it. Assarut et al. [4] evaluate the performance of Minn and Siu's DCA algorithm, and compare it with other schemes. Moreover, they propose a different algorithm for a more restricted setting [3]. Others use additional mechanisms like time multiplexing or code sharing on top of the original problem setting in order to mitigate the code-blocking problem [6], [23]. A different direction is to use a heuristic approach that solves the problem for small input instances [6]. Dell'Amico et al. [10] present a dynamic tree partitioning technique and evaluate it in simulations with respect to blocking probability and number of reassignments over a sequence of call arrivals and departures. Kam et al. [19] address the problem in the context of bursty traffic and different QoS (Quality of Service). They come up with a notion of "fairness" and also propose using multiplexing. Priority-based schemes for different QoS classes can be found in [9], similar in perspective are [14] and [15].

Fantacci and Nannicini [12] are among the first to express the problem in its online version, although they have quite a different focus. They come up with a scheme that is similar to the compact-representation scheme in Section 7, without focusing on the number of reassignments. Rouskas and Skoutas [23] propose a greedy online-algorithm that minimizes in each step the number of additionally blocked codes, and provide simulation results but no analysis. Chen and Chen [7] propose a best-fit least-recently used approach, also without analysis.

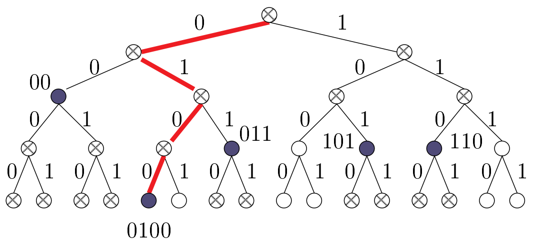


Fig. 2.1. Correspondence of code assignments in tree of height 4 with codes on levels {0, 1, 1, 1, 2} and prefix free codes of lengths {4, 3, 3, 3, 2}.

2. Observations on the CA Problem

2.1. *Feasibility of Code Assignment.* Given a tree T with n assigned codes at levels $l_1, \dots, l_n$ and a code insertion c for level l_{n+1}, we examine the existence of the code reassignment to insert code c. Clearly, this can be done if there exists a code assignment for desired levels $l_1, \dots, l_{n+1}$. Every assigned code on level l has its unique path from the root to a node of length $h - l$. The path can be encoded by a word $w \in \{0, 1\}^{h-l}$ determining whether we traverse through the left or right child. From the properties of code assignments the path/node identifiers form a binary prefix-free code. On the other hand, given a prefix-free code set of lengths $\{h - l_1, \dots, h - l_{n+1}\}$ we can clearly assign codes on levels l_i by following the paths described by the code words (see Figure 2.1). Thus, we have showed that a code assignment for codes on levels $l_1, \dots, l_{n+1}$ exists if and only if there exists a binary prefix-free code set of given lengths $\{h - l_1, \dots, h - l_{n+1}\}$.

Now we are ready to use the Kraft–McMillan inequality.

THEOREM 2.1. *A binary prefix-free code of code lengths $a_1, \dots, a_m$ exists if and only if*

$$\sum_{i=1}^{m} 2^{-a_i} \leq 1. \tag{2.1}$$

PROOF. For example, in [1]. □

By multiplying (2.1) by 2^h we immediately get the following corollary.

COROLLARY 2.2. *A code assignment for desired levels $l_1, \dots, l_m$ into the tree T of height h with N leaves exists if and only if*

$$\sum_{i=1}^{m} 2^{l_i} \leq N.$$

We see that checking whether we can successfully serve the code insertions can be done in linear time. Therefore, from now on we assume that the insertions always fit in the tree capacity, i.e. there exists a code reassignment to insert the code.

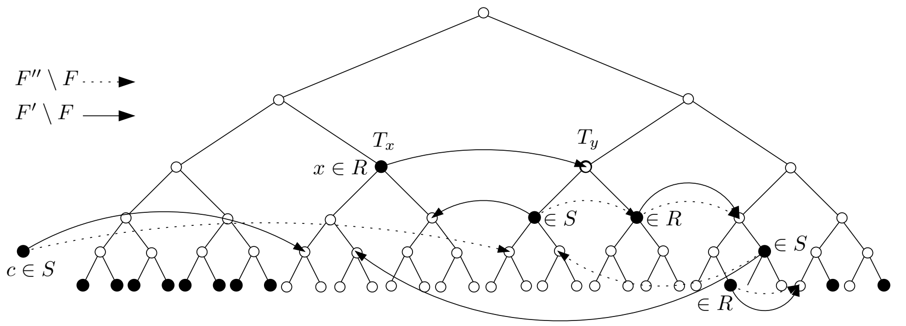


Fig. 2.2. Non-optimality of a code assignment F' that reassigns codes also on higher levels than the requested level.

2.2. *Irrelevance of Higher Level Codes.* In this section we show that an optimal algorithm for the one-step CA problem moves only codes on levels lower than the requested level r. A similar result was already given in [21], but here we give an independent and slightly different statement.

LEMMA 2.3. *Let c be an insertion on level r into a code tree T. Then for every code reassignment F' that inserts c and that moves a code on level $l \geq r$ there exists a code reassignment F'' that inserts c and moves fewer codes*, *i.e. with $|F'' \backslash F| < |F' \backslash F|$.*

PROOF. Let $x \in F$ be the highest code that is reassigned by F' on a level at or above the level r and let S denote the set of codes moved by F' into the subtree T_x rooted at node x. We denote by R the rest of the codes that are moved by F' (see Figure 2.2). The cost of F' is $|S| + |R|$. The code reassignment F'' is defined as follows: Let y be the position where F' moves the code x. Then F'' moves the codes in S into the subtree T_y rooted at y, leaves the code x at the root of T_x and moves the rest of the codes R in the same way as F'. The cost of F'' is at least one less than the cost of F' since it does not move the code x. In the example from Figure 2.2 the cost of F' is 6 and the cost of F'' is 5. $\square$

3. Non-Optimality of Greedy Algorithms. Here we look at possible greedy algorithms for the one-step offline CA. A straightforward greedy approach is to select for a code insertion a subtree with minimum cost that is not blocked by a code above the requested level, according to some cost function. All codes in the selected subtree must then be reassigned. So in every step a top-down greedy algorithm chooses the maximum bandwidth code that has to be reassigned, places it at the root of a minimum cost subtree, takes out the codes in that subtree and proceeds recursively. The DCA algorithm in [21] works in this way. The authors propose different cost functions, among which the "topology search" cost function is claimed to solve the one-step offline CA optimally. Here we show the following theorem:

THEOREM 3.1. *Any top-down greedy algorithm A_{tdg} depending only on the current assignment of the considered subtree is not optimal.*

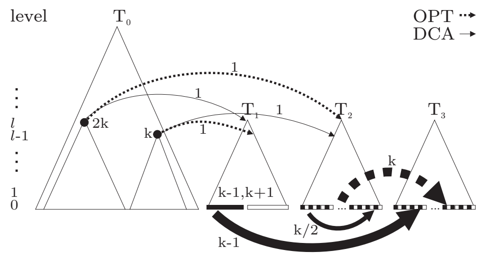


Fig. 3.1. Example for the proof of Theorem 3.1.

As all proposed cost functions in [21] depend only on the current assignment of the considered subtree, this theorem implies the non-optimality of the DCA algorithm.

PROOF. Our construction considers the subtrees in Figure 3.1 and the assignment of a new code to the root of the tree T_0. The tree T_0 has a code with bandwidth $2k$ on level l. Depending on the cost function, it can have in addition a code with bandwidth k on level $l - 1$. The subtree T_1 contains $k - 1$ codes at leaf level and the rest of the subtree is empty. The subtrees T_2 and T_3 contain k codes at leaf level interleaved with k free leaves. All other subtrees, in particular, the sibling trees of T_1, T_2 and T_3 (omitted from the figure) have all the leaves assigned. This pairing rules out all cost functions that do not put the initial code at the root of T_0. We are left with two cases:

Case 1: *The cost function evaluates T_2 and T_3 as cheaper than T_1.* In this case we let the subtree T_0 contain only the code with bandwidth $2k$. Algorithm A_{tdg} reassigns the code with bandwidth $2k$ to the root of the subtree T_2 or T_3, which causes one more reassignment than assigning it to the root of T_1, hence the algorithm fails to produce the optimal solution.

Case 2: *The cost function evaluates T_1 as cheaper than T_2 and T_3.* In this case we let the subtree T_0 have both codes. A_{tdg} moves the code with bandwidth $2k$ to the root of T_1 and the code with bandwidth k into the tree T_2 or T_3, see the solid lines in Figure 3.1. The number of reassigned codes is $3k/2+2$. However, the minimum number of reassignments is $k + 3$, which is achieved when the code with bandwidth k is moved in the empty part of T_1 and the code with bandwidth $2k$ is moved to the root of T_2 or T_3, see the dashed lines in Figure 3.1. □

4. *NP*-Hardness of One-Step Offline CA.

Here we prove the decision variant of the one-step offline CA to be *NP*-complete. The (canonical) decision variant of it is to decide whether a new code insertion can be handled with cost less than or equal to a number $c_{\max}$, which is also part of the input. First, we note that the decision variant is in *NP*, because we can guess an optimal assignment and verify in polynomial time, if it is feasible and if its cost is lower than or equal to $c_{\max}$. Now the *NP*-completeness is established by a reduction from the three-dimensional matching problem (3DM) that we restate here for completeness (see [16]):

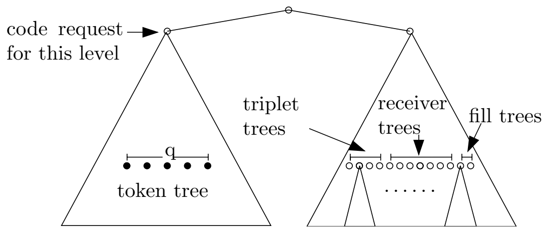


Fig. 4.1. Sketch of the construction.

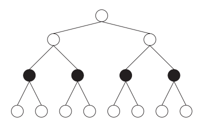
PROBLEM 4.1 (3DM). Given a set $M \subseteq W \times X \times Y$, where W, X and Y are disjoint sets having the same number q of elements. Does M contain a perfect matching, i.e. a subset $M' \subseteq M$ such that $|M'| = q$ and no two elements of M' agree in any coordinate?

Let us index the elements of the ground sets W, X, Y from 1 to q. To simplify the presentation, we introduce the *indicator vector* of a triplet (w_i, x_j, y_k) as a zero–one vector of length $3q$ that is all zero except at the indices $i, q + j$ and $2q + k$. The idea of the reduction is to view the triplets as such indicator vectors and to observe that the problem 3DM is equivalent to finding a subset of q indicator vectors out of the indicator vectors in M that sum up to the all-one vector.

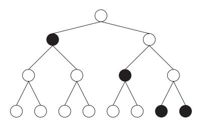
Figure 4.1 shows an outline of the construction that we use for the reduction. An input to 3DM is transformed into an initial feasible assignment that consists of a token tree on the left side and different smaller trees on the right. A code insertion request is given at the level indicated in the figure. The construction is set up in such a way that the code must be assigned to the root of the left tree, the *token tree*, in order to minimize the number of reassignments. For the same reason, the q codes that are forced to move from the left to the right tree must be assigned to the roots of *triplet trees*. The choice of the q triplet trees reflects the choice of the corresponding triplets of a matching. All codes in the chosen triplet trees find a place without any additional reassignment if and only if these triplets really represent a 3D matching.

Let us now look into the details of the construction. The token tree consists of q codes positioned arbitrarily on level l_{start} with sufficient depth, for example, depth $\lceil \log(|M| + 21q^2 + q) \rceil + 1$. The triplet trees have their roots on the same level l_{start}. They are constructed from the indicator vectors of the triplets. For each of the $3q$ positions of the vector such a tree has four levels—together called a *layer*—that encode either zero or one, where the encodings of zero and one are shown in Figure 4.2(a) and (b). Figure 4.2(c) and (d) shows how layers are stacked using *sibling trees* (the sibling tree of a zero-tree is identical to that of a one-tree shown in the figure). We have chosen the zero-trees and one-trees such that both have the same number of codes and occupy the same bandwidth, but are still different.

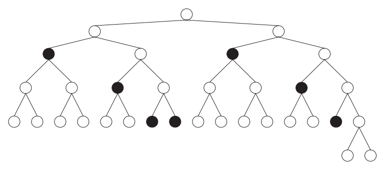
The receiver trees are supposed to receive all codes in the chosen triplet trees. These codes fit exactly in the free positions, if and only if the chosen triplets form a 3DM, i.e. if their indicator vectors sum up to the all-one vector. This equivalence directly tells us, how many codes the trees must receive on which level: On every layer the receiver trees must take $q - 1$ zero-trees, 1 one-tree and q sibling-trees, so that on the four levels of



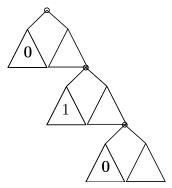
(a) The zero-tree



(b) The one-tree



(c) A layer, consisting of a one-tree and its sibling



(d) Stacking layers

Fig. 4.2. Encoding of zero and one.

each layer there must be exactly $0, q+1, 5q-3$, resp. $q+2$, free codes (plus q extra codes on the very last level). For each one of these $3q \cdot 7q + q = 21q^2 + q$ codes we build one receiver tree. The receiver tree for a code on level l' is a tree with root on level l_{start} with the following properties. It has one free position on level l', the rest of the tree is full and it contains $21q + 2$ codes, i.e. one more code than a triplet tree. Clearly, such a tree always exists in our situation.

Finally, the fill trees are trees that are completely full and have one more code than the receiver trees. They fill up the level l_{start} in the sibling-tree of the token tree.

An interesting question is, whether this transformation from 3DM to the one-step offline CA can be done in polynomial time. This depends on the input encoding of our problem. We consider the following natural encodings:

- A zero–one vector that specifies for every node of the tree whether there is a code or not.
- A sparse representation of the tree, consisting only of the positions of the assigned codes.

Obviously, the transformation cannot be done in polynomial time for the first input encoding, because the generated tree has $2^{12q+l_{\text{start}}}$ leaves. For the second input encoding the transformation is polynomial, because the total number of generated codes is polynomial in q, which is polynomial in the input size of 3DM. Besides, we should rather not expect an *NP*-completeness proof for the first input encoding, because this would suggest—together with the dynamic programming algorithm in this paper—$n^{O(\log n)}$-algorithms for all problems in *NP*.

We now state the crucial property of the construction in a lemma:

LEMMA 4.2. *Let M be an input for $3DM$ and let φ be the transformation described above. Then $M \in 3DM$ if and only if $\varphi(M)$ can be done with $\alpha = 21q^2 + 2q + 1$ reassignments.*

PROOF. Assume there is a 3DM $M' \subset M$. Now consider the reassignment that assigns the code insertion to the root of the token tree, and the tokens to the q roots of the triplet trees that correspond to the triplets in M'. We know that the corresponding indicator vectors sum up to the all-one vector, so that all codes in the triplet trees that need to be reassigned fit exactly in the receiver trees. In total, $1 + q + (21q + 1)q = \alpha$ codes are (re-)assigned.

Now assume there is no matching. This implies that every subset of q indicator vectors does not sum up to the all-one vector. Assume for a contradiction that we can still serve $\varphi(M)$ with at most α reassignments. Clearly, the initial code insertion must be assigned to the left tree, otherwise we need too many reassignments. The q tokens must not trigger more than $(21q + 1)q$ additional reassignments. This is only possible if they are all assigned to triplet trees, which triggers exactly $(21q + 1)q$ necessary reassignments. Now no more reassignments are allowed. However, we know that the corresponding q indicator vectors do not sum up to the all-one vector, in particular, there must be one position that sums up to zero. In the layer of this position the receiver-trees receive q zero-trees and no one-tree instead of $q - 1$ zero-trees and one one-tree. However, by construction the extra zero-tree cannot be assigned to the remaining receiver trees of the one-tree. It cannot be assigned somewhere else either, because this would cause an extra reassignment on a different layer. This is why an extra reassignment is needed, which brings the total number of (re-)assignments above α. □

One could wonder whether an optimal one-step offline CA algorithm can ever attain the configuration that we construct for the transformation. We prove in the next section in Corollary 4.5 that we can force such an algorithm into any configuration. To sum up, we have shown the following theorem:

THEOREM 4.3. *The decision variant of the one-step offline CA is NP-complete for an input given by a list of positions of the assigned codes and the code insertion level.*

4.1. *Enforcing Arbitrary Configurations.*

In this section we show that for any configuration C' and any optimal one-step algorithm A there exists a sequence of code insertions and deletions of polynomial length, so that A ends up in C' on that sequence. Notice that any optimal one-step algorithm reassigns codes only if it has to, i.e. it places a code without any additional reassignments if this is possible, and it does not reassign after a deletion. The result even applies to any algorithm A with these properties.

We start with the empty configuration C_0. The idea of the proof is to take a detour and first attain a full-capacity configuration C_{full} and then go from there to C'. The second step is easy: It suffices to delete all the codes in C_{full} that are not in C'; A must not do any reassignments during these deletions. First, we show that we can force A to produce an arbitrarily chosen configuration C_{full} that uses the full tree capacity.

THEOREM 4.4. *Any one-step optimal algorithm A can be led to an arbitrary full configuration C_{full} with n assigned codes by a request sequence of length $m < 3n$.*

PROOF. Recall that h denotes the height of the code tree. We proceed top-down: On every level l' with codes in C_{full} we first fill all its unblocked positions using at most

$2^{h-l'}$ code insertions on level l'. A just fills l' with codes. Then we delete all codes on l' that are not in C_{full} and proceed recursively to the next level.

Now we have to argue that we do not insert too many codes in this process. To see this, observe that we are only inserting and deleting codes above the n codes in C_{full}, and we do this at most once in every node. Now if we consider the binary tree, the leaves of which are the codes in C_{full}, then we see that the number of insert operations is bounded by $n + n - 1$, where $n - 1$ is the number of inner nodes of this tree. Together with the deletions we obtain the statement. □

We return to arbitrary configurations.

COROLLARY 4.5. *Given a configuration tree C' of height h with n assigned codes, there exists a sequence $\sigma_1, \ldots, \sigma_m$ of code insertions and deletions of length $m < 4nh$ that forces A into C'.*

PROOF. We define C_{full} from C' by filling the gap trees in C' (as high as possible) with codes. Each code causes at most one gap tree on every level, hence we need at most h codes to fill the gap trees for one code. Altogether we need at most nh codes to fill all gap trees. According to Theorem 4.4, we can construct a sequence of length $m < 3nh$ that forces A into C_{full}. Then we delete the padding codes and end up in C'. Altogether we need at most $4nh$ requests for code insertion and deletion. □

5. Exact $n^{\mathcal{O}(h)}$ Dynamic Programming Algorithm.

In this section we solve the one-step offline CA problem optimally using a dynamic programming approach. The key idea of the resulting algorithm is to store the right information in the nodes of the tree and to build it up in a bottom-up fashion.

To make this construction precise, we define a *signature* of a subtree T_v with root v as an $(l(v) + 1)$-dimensional vector $s^v = (s_0^v, \ldots, s_{l(v)}^v)$, in which s_i^v is the number of codes in T_v on level i. A signature s is *feasible* if there exists a subtree T_v with a code assignment that has signature s. The information stored in every node v of the tree consists of a table, in which all possible feasible signatures of an arbitrary tree of height $l(v)$ are stored together with their *cost for* T_v. Here the cost of such a signature s for T_v (usually $s \neq s^v$) is defined as the minimum number of codes in T_v that have to move away from their old position in order to attain some tree T_v' with signature s. To attain T_v' it can be necessary to move also into T_v codes from other subtrees but we do not count these movements for the cost of s for T_v.

Given a code tree T with all these tables computed, one can compute the cost of any single code insertion from the table at the root node r: Let $s^r = (s_0^r, \ldots, s_h^r)$ be the signature of the whole code tree before insertion, then the cost of an insertion at level l is the cost of the signature $(s_0^r, \ldots, s_l^r + 1, \ldots, s_h^r)$ in this table plus one. This follows because the minimum number of codes that are moved away from their positions in T is equal to the number of reassignments minus one.

The computation of the tables starts at the leaf level, where the cost of the one-dimensional signatures is trivially defined. At any node v of level $l(v)$ the cost $c(v, s)$ of signature s for T_v is computed from the cost incurred in the left subtree T_l of v plus the cost incurred in the right subtree T_r plus the cost at v. The costs $c(l, s')$ and $c(r, s'')$ in the subtrees come from two feasible signatures with the property $s = (s'_0 + s''_0, \ldots, s'_{l(v)-1} + s''_{l(v)-1}, s_{l(v)})$. Any pair (s', s'') of such signatures corresponds to a possible configuration after the code insertion. The best pair for node v gives $c(v, s)$. Let $s^v = (s^v_0, \ldots, s^v_{l(v)})$ be the signature of T_v, then it holds that

$$c(v, s) = \begin{cases} c(l, (0, \ldots, 0)) + c(r, (0, \ldots, 0)) & \text{for} \quad s_{l(v)} = 1, \\ \min_{\{s', s'' \mid (s', 0) + (s'', 0) = s\}} (c(l, s') + c(r, s'')) & \text{for} \quad s_{l(v)} = 0, \quad s^v_{l(v)} = 0, \\ 1 & \text{for} \quad s_{l(v)} = 0, \quad s^v_{l(v)} = 1. \end{cases}$$

The costs of all signatures s for v can be calculated simultaneously by combining the two tables in the left and right children of v. Observe for the running time that the number of feasible signatures is bounded by $(n + 1)^h$ because there cannot be more than n codes on any level. The time to combine two tables is $\mathcal{O}(n^{2h})$, thus the total running time is bounded by $\mathcal{O}(2^h \cdot n^{2h})$.

THEOREM 5.1. *The one-step offline CA can be optimally solved in time $\mathcal{O}(2^h \cdot n^{2h})$ and space $\mathcal{O}(h \cdot n^h)$.*

6. An *h*-Approximation Algorithm for One-Step Offline CA.

In this section we propose and analyze a greedy algorithm for one-step offline CA, i.e. for the problem of assigning an initial code insertion c_0 into a code tree T with given code assignment F. The idea of the greedy algorithm A_{greedy} is to assign the code c_0 onto the root g of the subtree T_g that contains the fewest assigned codes among all possible subtrees. From Lemma 2.3 we know that no optimal algorithm reassigns codes on higher levels than the current one; hence the possible subtrees are those that do not contain assigned codes on or above their root. Then the greedy algorithm takes all codes in T_g (denoted by $\Gamma(T_g)$) and reassigns them recursively in the same way, always processing codes of higher level first.

At every time t algorithm A_{greedy} has to assign a set C_t of codes into the current tree T^t. Initially, $C_0 = \{c_0\}$ and $T^0 = T$. Recall that for a given position, code or code insertion c, its level is denoted by $l(c)$.

ALGORITHM 6.1 (Greedy Algorithm A_{greedy}).

```
$C_0 \leftarrow \{c_0\}$;  $T^0 \leftarrow T$
$t \leftarrow 0$
WHILE $C_t \neq \emptyset$ DO
  $c_t \leftarrow$ element with highest level in $C_t$
  $g \leftarrow$ the root of a subtree $T^t_g$ of level $l(c_t)$ with the fewest
    codes in it and no code on or above its root
  /* assign $c_t$ to position $g$ */
  $T^{t+1} \leftarrow (T^t \backslash \Gamma(T^t_g)) \cup \{g\}$
  $C_{t+1} \leftarrow (C_t \cup \Gamma(T^t_g)) \backslash \{c_t\}$
  $t \leftarrow t + 1$
END WHILE
```

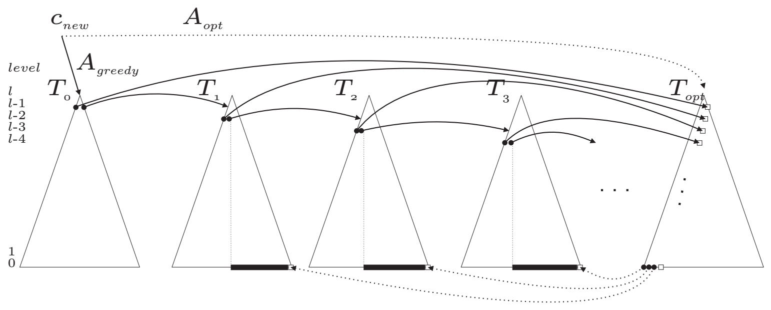


Fig. 6.1. Example for the lower bound for A_{greedy}.

In [21] a similar algorithm is proposed as a heuristic for the one-step offline CA. We prove that A_{greedy} has approximation ratio h. This bound is asymptotically tight: In the following examples we show that A_{greedy} can be forced to use $\Omega(h) \cdot OPT$ (re-)assignments (see Figure 6.1), where OPT refers to the optimal number of (re-)assignments. A new code c_{new} is assigned by A_{greedy} into the root of T_0 (which contains the least number of codes). The two codes on level $l-1$ from T_0 are reassigned as shown in the figure, one code can be reassigned into T_{opt} and the other one goes recursively into T_1. In total, A_{greedy} does $2 \cdot l + 1$ (re-)assignments while the optimal algorithm assigns c_{new} into the root of T_{opt} and reassigns the three codes from the leaf level into the trees T_1, T_2, T_3, requiring only four (re-)assignments. Obviously, for this example A_{greedy} is not better than $(2l+1)/4$ times the optimal. In general l can be $\Omega(h)$.

For the upper bound we compare A_{greedy} with the optimal algorithm A_{opt}. A_{opt} assigns c_0 to the root of a subtree T_{x_0}, the codes from T_{x_0} to some other subtrees, and so on. Let us call the set of subtrees to the root of which A_{opt} moves codes the *opt-trees*, denoted by $\mathcal{T}_{opt}$, and the arcs that show how A_{opt} moves the codes the *opt-arcs* (see Figure 6.2). By $V(\mathcal{T}_{opt})$ we denote the set of nodes in $\mathcal{T}_{opt}$.

A sketch of the proof is as follows. First, we show that in every step t, A_{greedy} has the possibility of assigning the codes in C_t into positions inside the opt-trees. This possibility can be expressed by a code mapping $\varphi_t : C_t \to V(\mathcal{T}_{opt})$. The key property is now that in every step of the algorithm there is the theoretical choice of completing the current assignment using the code mapping φ_t and the opt-arcs as follows: Use φ_t to assign the codes in C_t into positions in the opt-trees and then use the opt-arcs to move codes out of these subtrees of the opt-trees to produce a feasible code assignment. We will see that

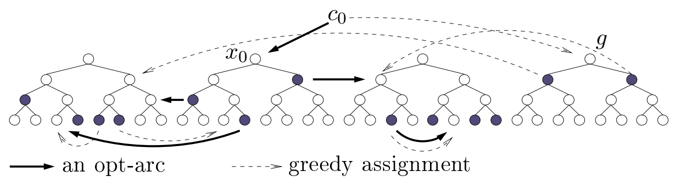


Fig. 6.2. A_{opt} moves codes to assign a new code c_0 using opt-arcs. The opt-trees are subtrees to the root of which A_{opt} moves codes. Here, the cost of the optimal solution is 5. The greedy algorithm has cost 6.

this property is enough to ensure that A_{greedy} incurs a cost of no more than OPT on every level.

In the process of the algorithm it can happen that we have to change the opt-arcs in order to ensure the existence of φ_t. To model the necessary changes we introduce α_t-arcs that represent the changed opt-arcs after t steps of the greedy algorithm.

To make the proof-sketch precise, we need the following definitions:

DEFINITION 6.2. Let $\mathcal{T}_{opt}$ be the set of the opt-trees for a code insertion c_0 and let T^t (together with its code assignment F^t) be the code tree after t steps of the greedy algorithm A_{greedy}. An *α-mapping* at time t is a mapping $\alpha_t : M_{\alpha_t} \to V(\mathcal{T}_{opt})$ for some $M_{\alpha_t} \subseteq F^t$, such that, $\forall v \in M_{\alpha_t}$, $l(v) = l(\alpha_t(v))$ and $\alpha_t(M_{\alpha_t}) \cup (F^t \backslash M_{\alpha_t})$ is a code assignment.

Note that in general F^t is not a code assignment for all codes since it does not contain the codes in C^t. The set $\alpha_t(M_{\alpha_t}) \cup (F^t \backslash M_{\alpha_t})$ represents the resulting code assignment (that again does not contain the codes in C^t) after reassignment of the codes $M_{\alpha_t} \subseteq F^t$ by α_t.

DEFINITION 6.3. Let T^t be a code tree, let x and y be positions in T^t and let α_t be an α-mapping. We say that y *depends* on x in T^t and α_t if there is a path from x to y using only tree-edges from a parent to a child and α_t-arcs. By $\mathrm{dep}_t(x)$ we denote the set of all positions y that depend on x in T^t and α_t. We say that an α_t-arc (u, v) depends on x if $u \in \mathrm{dep}_t(x)$.

For an illustration of this definition, see Figure 6.3.

DEFINITION 6.4. At time t a pair (φ_t, α_t) of a code mapping $\varphi_t : C_t \to V(\mathcal{T}_{opt})$ and an α-mapping α_t is called an *independent mapping* for T^t, if the following properties hold:

1. $\forall c \in C_t$ the levels of $\varphi_t(c)$ and c are the same (i.e. $l(c) = l(\varphi_t(c))$).
2. $\forall c \in C_t$ there is no code in T^t at or above the roots of the trees in $\mathrm{dep}_t(\varphi_t(c))$.
3. The code movements realized by φ_t and α_t (i.e. the set $\varphi_t(C_t) \cup \alpha_t(M_{\alpha_t}) \cup (F^t \backslash M_{\alpha_t})$) form a code assignment.
4. Every node in the domain M_{α_t} of α_t is contained in $\mathrm{dep}_t(\varphi_t(C_t))$ (i.e. no unnecessary arcs are in α_t).

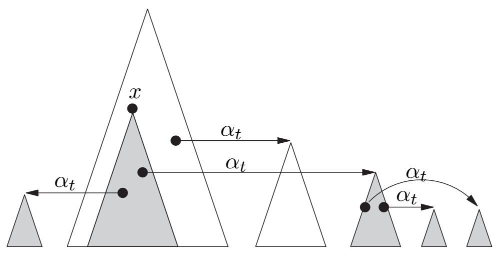


Fig. 6.3. The filled subtrees represent all the positions that depend on x.

Note that φ_t and α_t can be viewed equivalently as functions and as collections of arcs of the form $(c, \varphi_t(c))$ and $(u, \alpha_t(u))$, respectively. We write $\text{dep}_t(\varphi_t(C_t))$ for the set $\bigcup_{c\in C_t} \text{dep}_t(\varphi_t(c))$. Note that if a pair (φ_t, α_t) is an independent mapping for T^t, then $\text{dep}_t(\varphi_t(C_t))$ is contained in opt-trees and every node in $\text{dep}_t(\varphi_t(C_t))$ can be reached on exactly one path from C_t (using one φ_t-arc and an arbitrary sequence of tree-arcs, which always go from parent to child, and α_t-arcs from a code $c \in \Gamma(T^t)$ to $\alpha_t(c)$).

Now we state a lemma that is crucial for the analysis of the greedy strategy, the proof of which we give in Section 6.1.

LEMMA 6.5. *For every set C_t in algorithm A_{greedy} the following invariant holds*:

(6.1) There is an independent mapping $(\varphi_t, \alpha_t)$for T^t.

We remark that Lemma 6.5 actually applies to all algorithms that work level-wise top-down and choose a subtree T_g^t for each code $c_t \in C_t$ arbitrarily under the condition that there is no code on or above the position g.

We can express the cost of the optimal solution by the opt-trees:

LEMMA 6.6. (a) *The optimal cost is equal to the number of assigned codes in the opt-trees plus one*, *and* (b) *it is equal to the number of opt-trees*.

PROOF. Observe for (a) that A_{opt} moves all the codes in the opt-trees and for (b) that A_{opt} moves one code into the root of every opt-tree. □

THEOREM 6.7. *The algorithm A_{greedy} has an approximation ratio of h.*

PROOF. A_{greedy} works level-wise top-down. We show that on every level l the greedy algorithm incurs a cost of at most OPT. Consider a time t_l where A_{greedy} is about to start a new level l, i.e. before A_{greedy} assigns the first code on level l. Assume that C_{t_l} contains q_l codes on level l. Then A_{greedy} places these q_l codes in the roots of the q_l subtrees on level l containing the fewest codes. The code mapping φ_{t_l} that is part of the independent mapping $(\varphi_{t_l}, \alpha_{t_l})$, which exists by Lemma 6.5, maps each of these q_l codes to a different position in the opt-trees. Therefore, the total number of codes in the q_l subtrees with roots at $\varphi_{t_l}(c)$ (for c a code on level l in C_{t_l}) is at least the number of codes in the q_l subtrees chosen by A_{greedy}. Combining this with Lemma 6.6(a), we see that on every level A_{greedy} incurs a cost (number of codes that are moved away from their position in the tree) that is at most A_{opt}'s total cost. □

6.1. *Proof of Lemma* 6.5. We prove the lemma by induction on t. Assume that the code c_0 is to be inserted into the tree initially, and that A_{opt} assigns it to position x_0. For the base of the induction ($t = 0$), let $\varphi_0(c_0) = x_0$ and let α_0 consist of all opt-arcs, i.e. all arcs (u, v) such that A_{opt} moves a code from u to v. It is easy to see that (φ_0, α_0) is an independent mapping.

Now let $t \geq 0$ and assume that the lemma holds after t iterations of the greedy algorithm. We show how to construct $(\varphi_{t+1}, \alpha_{t+1})$ from the independent mapping (φ_t, α_t).

In iteration $t+1$, the greedy algorithm A_{greedy} assigns the code c_t of highest level in C_t to a feasible position g in T^t.

Case 1: *There is a code c_t' in C_t with $\varphi_t(c_t') = g$.* If $c_t' \neq c_t$, we exchange the φ_t values of c_t' and c_t while maintaining (φ_t, α_t) as an independent mapping for T^t. Thus, we can assume that $\varphi_t(c_t) = g$. We set

$$\varphi_{t+1} = \{(c, \varphi_t(c)) \mid c \in C_t \backslash \{c_t\}\} \cup \{(c, \alpha_t(c)) \mid c \in \Gamma(T_g^t)\}$$

and

$$\alpha_{t+1} = \alpha_t \backslash \{(c, \alpha_t(c)) \mid c \in \Gamma(T_g^t)\}.$$

It is easy to see that $(\varphi_{t+1}, \alpha_{t+1})$ is an independent mapping for T^{t+1}.

We remark that Case 1 could also be handled in the same way as Case 2 below, but we have chosen to give a direct treatment of Case 1 in order to illustrate some of the proof ideas on a simple case.

Case 2: *There is no code c_t' in C_t with $\varphi_t(c_t') = g$.* In this case, T_g^t can contain a number of codes, some of which may be in the domain M_{α_t} of α_t. Furthermore, there can be φ_t-arcs and α_t-arcs pointing into T_g^t. An example is shown in Figure 6.4. We have to define a φ_{t+1}-arc for all codes in T_g^t, and we must find a new destination outside T_g^t for those φ_t-arcs and α_t-arcs pointing into T_g^t that we need for the construction of $(\varphi_{t+1}, \alpha_{t+1})$.

First we will define an intermediate *generalized* independent mapping (φ, α) for T^{t+1} in which we allow *loose ends*, i.e. we allow a code c to have as head of its α-arc or φ-arc a *dummy tree* (that is not part of the real tree) of the required capacity. In a second step we will fix loose ends by finding proper destinations in $\text{dep}(\varphi_t(c_t))$ for them (where dep refers to the dependency induced by tree-arcs and the current α-arcs). In the end a part of the resulting (φ, α) without loose ends will be used to define $(\varphi_{t+1}, \alpha_{t+1})$.

We proceed as follows. For each assigned code c at a node v in T_g^t that is not in the domain M_{α_t} of α_t, define $\varphi(c) = v$. For each assigned code c in T_g^t that has an α_t-arc, define $\varphi(c) = \alpha_t(c)$. For all codes c in $C_t \backslash \{c_t\}$, set $\varphi(c) = \varphi_t(c)$. Let $\alpha = \alpha_t \backslash \{(u, v) \mid u \in T_g^t\}$. Finally, replace each of the just defined α-arcs or φ-arcs (u, v) for which

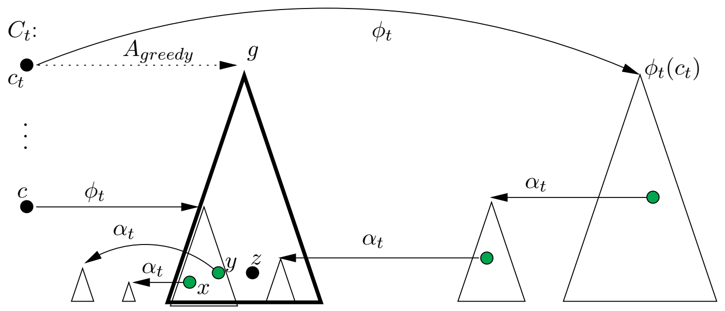


Fig. 6.4. T_g^t contains the heads of α_t-arcs and φ_t-arcs as well as codes with and without α_t-arcs.

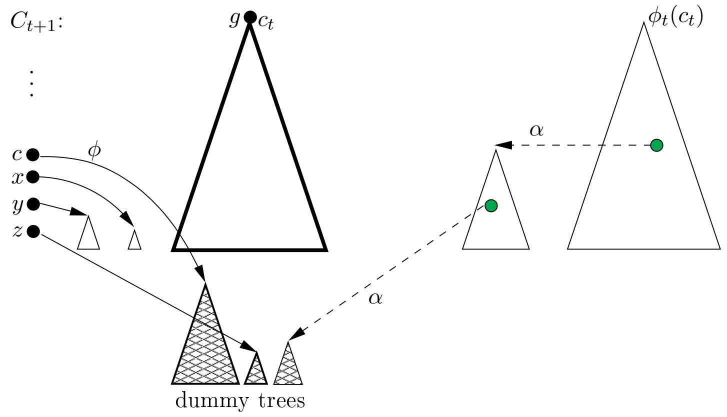


Fig. 6.5. The constructed generalized independent mapping (φ, α). All shown α-arcs are inactive (indicated by dashed lines). The rightmost dummy tree is inactive, the other two are active.

$v \in T_g^t$ by a loose end, i.e. an α-arc or φ-arc pointing from u to a dummy tree of height $l(v)$. The mapping (φ, α) constructed in this way is indeed a generalized independent mapping. Figure 6.5 shows the generalized mapping (φ, α) resulting from the situation in Figure 6.4.

Dummy trees that can be reached from $\varphi_t(c_t)$ along tree-arcs and α-arcs are called *inactive*, all other dummy trees are called *active*. Active dummy trees have to be fixed (so that we can eventually obtain an independent mapping without loose ends), while inactive dummy trees will either become active later or will be discarded in the end. Similarly, we call all α-arcs that can be reached from $\varphi_t(c_t)$ along tree-arcs and α-arcs *inactive*, and all other α-arcs *active*. Inactive α-arcs will either become active later or will be discarded in the end as well.

The active dummy trees will be placed step by step and the functions φ and α are updated after each step. Only inactive trees that are smaller than the assigned active tree can become active.

Let U denote the capacity of the tree T_g, i.e. $U = 2^{l(g)}$. Note that all dummy trees were generated from independent subtrees of T_g^t. Therefore, the total capacity of all dummy trees is at most U. Let U_a be the total capacity of active dummy trees and let U_i be the total capacity of inactive dummy trees. We have $U_a + U_i \leq U$.

We want to use dep$(\varphi_t(c_t))$ for finding new destinations for α-arcs or φ-arcs that point to dummy trees. We say that a path from $\varphi_t(c_t)$ to some tree node v is *strict* if it follows tree-arcs downward from nodes without assigned codes in T^{t+1} and α-arcs from nodes with assigned codes in T^{t+1}. Now we can define the *available capacity* in dep$(\varphi_t(c_t))$ to be the number of leaves that are not in dummy trees and that can be reached from $\varphi_t(c_t)$ along a strict path that does not contain the head of any φ-arc or active α-arc. Note that a position v in dep$(\varphi_t(c_t))$ can be used as the new head of an α-arc or φ-arc if and only if v is not in a dummy tree, there is no code at or above v, and no φ-arc or active α-arc points to a position in dep(v) or to a position p such that v is in dep(p). Otherwise, the position v is called *unavailable*.

The available capacity in $\mathrm{dep}(\varphi_t(c_t))$ is $U - U_i$ initially, since only the loose ends in $\mathrm{dep}(\varphi_t(c_t))$ reduce the available capacity. The total capacity of active dummy trees is $U_a \leq U - U_i$. In the following we will maintain the invariant that the total capacity of active dummy trees is at most the available capacity in $\mathrm{dep}(\varphi_t(c_t))$.

We fix the active dummy trees one by one in order of non-increasing levels. Assume that we are currently processing a dummy tree of level d that is the head of an α-arc or φ-arc (x, y). Consider all nodes v_d of level d in T^{t+1} that do not have assigned codes and are reachable from $\varphi_t(c_t)$ along strict paths. Observe that a node v_d is unavailable only if it is inside an inactive dummy tree or if the path from $\varphi_t(c_t)$ to v_d passes through the head of an active α-arc or a φ-arc. However, it is not possible that all nodes v_d are unavailable, because then the total available capacity in $\mathrm{dep}(\varphi_t(c_t))$ would be zero, contradicting our invariant. Thus, we can find a node v_d that is available (i.e. not unavailable). We replace (x, y) by (x, v_d) and make all α-arcs reachable from v_d as well as all inactive dummy trees reachable from v_d active. (Note that no active dummy tree can have been reachable from v_d before this operation, since we fix the active dummy trees in order of non-increasing levels.) Let U' be the total capacity of previously inactive dummy trees that were made active now. The total capacity of active dummy trees decreases by $2^d - U'$, and the total available capacity in $\mathrm{dep}(\varphi_t(c_t))$ decreases by $2^d - U'$ as well (since the part of $\mathrm{dep}(\varphi_t(c_t))$ that is reachable from v_d had available capacity exactly $2^d - U'$). Therefore, the invariant is maintained and the process can be continued until no active dummy trees are left. The process terminates because the total capacity of active dummy trees never increases and in each step the number of active dummy trees of the highest level decreases by one (and only dummy trees of lower levels may become active). A possible result of applying this process to the generalized independent mapping of Figure 6.5 is shown in Figure 6.6.

When all active dummy trees are fixed, we let $\varphi_{t+1} = \varphi$ and $\alpha_{t+1} = \{(u, v) \in \alpha \mid (u, v) \text{ is active }\}$. Since (φ, α) was a generalized independent mapping and $(\varphi_{t+1}, \alpha_{t+1})$ does not contain loose ends, we have that $(\varphi_{t+1}, \alpha_{t+1})$ is an independent mapping as required. $\square$

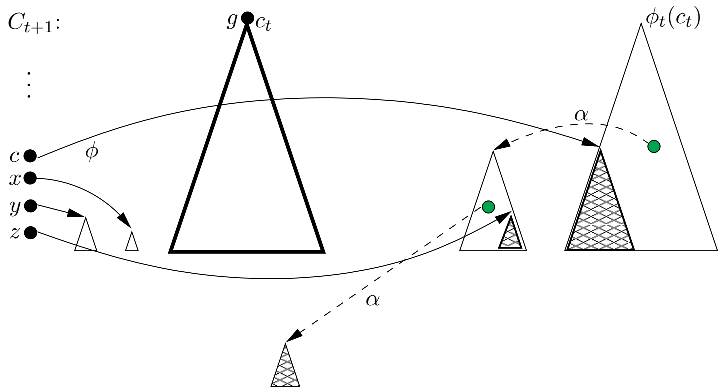


Fig. 6.6. The final generalized independent mapping (φ, α) in which all active dummy trees have been fixed. The independent mapping $(\varphi_{t+1}, \alpha_{t+1})$ is obtained by deleting the inactive α-arcs and discarding the remaining inactive dummy tree.

7. Online Code Assignment.

Here we study the CA problem in an online setting. We assume that insertions do not exceed the total available bandwidth.

In an *online problem* the input is received in an online manner and the output must be produced online [5], [13]. In the case of the online CA problem the requests for code insertions and deletions must be handled one after another, i.e. the ith request must be served before the $(i+1)$st request is known. An online algorithm ALG for the CA problem is *c-competitive* if there is a constant α such that, for all finite input sequences I,

$$\mathrm{ALG}(I) \leq c \cdot \mathrm{OPT}(I) + \alpha.$$

In this case the *competitive ratio* of ALG is c. It is common to think of the input sequence as a sequence being generated by a (malicious) *adversary*.

We give a lower bound on the competitive ratio, analyze several algorithms and present a resource-augmented algorithm with constant competitive ratio.

THEOREM 7.1. *No deterministic algorithm A for the online CA problem can be better than* 1.5*-competitive.*

PROOF. Let A be any deterministic algorithm for the problem. Consider N leaf insertions. The adversary can delete $N/2$ codes (every second) to get the situation in Figure 7.1.

Then a code insertion at level $h-1$ causes $N/4$ code reassignments. We can proceed with the left subtree of full leaf codes recursively and repeat this process $(\log_2 N - 1)$ times. The optimal algorithm A_{opt} assigns the leaves in the first step in such a way that it does not need any reassignment at all. Thus, A_{opt} needs $N + \log_2 N - 1$ code assignments. Algorithm A needs $N + T(N)$ code assignments, where $T(N) = 1 + N/4 + T(N/2)$ and $T(2) = 0$. Clearly, $T(N) = \log_2 N - 1 + (N/2)(1 - 2/N)$. If $C_A \leq c \cdot C_{OPT}$ then

$$c \;\geq\; \frac{3N/2 + \log_2 N - 2}{N + \log_2 N - 1} \longrightarrow_{N\to\infty} \frac{3}{2}. \qquad \square$$

7.1. *Compact Representation Algorithm.* This algorithm maintains the codes in the tree T sorted and compact. For a given node/code $v \in T$ we denote by $l(v)$ its level

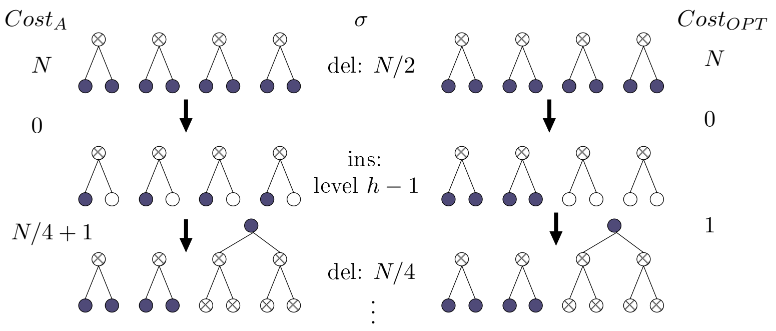


Fig. 7.1. Lower bound for the online assignment problem.

and by $w(v)$ its string representation, i.e. the description of the path from the root to the node/code, where 0 means a left child and 1 a right child. We use the lexicographic ordering when comparing two string representations. By U we denote the set of unblocked nodes of the tree. We maintain the following invariants:

(7.1) for all codes $u, v \in F,\ l(u) < l(v) \Rightarrow w(u) < w(v)$,

(7.2) for all nodes $u, v \in T,\ l(u) \leq l(v) \wedge u \in F \wedge v \in U \Rightarrow w(u) < w(v)$.

This states that we want to keep the codes in the tree ordered from left to right according to their levels (higher-level assigned codes are to the right of lower-level assigned codes) and compact (no unblocked code to the left of any assigned code on the same level).

In the following analysis we show that this algorithm is not worse than $\mathcal{O}(h)$ times the optimum for the offline version. We also give an example that shows that the algorithm is not asymptotically better than this.

THEOREM 7.2. *Algorithm A_{compact} satisfying invariants* (7.1) *and* (7.2) *performs at most h code reassignments per insertion or deletion.*

PROOF. We show that for both insertion and deletion we need to make at most h code reassignments. When inserting a code on level l, we look for the rightmost unassigned position on that level that maintains the invariants (7.1) and (7.2) among codes on level $0, \ldots, l$. Either the found node is not blocked, so that we do not move any codes, or the code is blocked by some assigned code on a higher level $l' > l$ (see Figure 7.2). In the latter case we remove this code to free the position for level l and handle the new code insertion on level l' recursively. Since we move at most one code at each level and we have h levels, we move at most h codes for each insertion.

Handling the deletion operation is similar, we just move the codes from right to left in the tree and move at most one code per level to maintain the invariants. □

COROLLARY 7.3. *Algorithm $A_{compact}$ satisfying invariants* (7.1) *and* (7.2) *is $\mathcal{O}(h)$-competitive.*

PROOF. In the sequence $\sigma = \sigma_1, \ldots, \sigma_m$ the number of deletions d must be smaller or equal to the number i of insertions, which implies $d \leq m/2$. The cost of any optimal algorithm is then at least $i \geq m/2$. On the other hand, $A_{compact}$ incurs a cost of at most $m \cdot h$, which implies that it is $\mathcal{O}(h)$-competitive. □

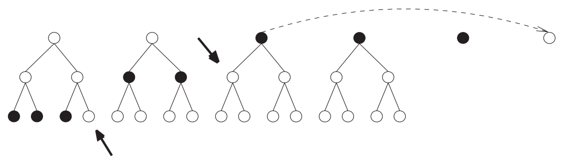


Fig. 7.2. For a code insertion, Algorithm $A_{compact}$ finds the leftmost position (blocked or unblocked) that has no code on it and no code in the subtree below it. It reassigns at most one code at every level.

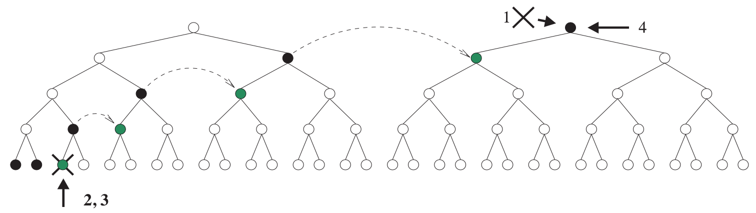


Fig. 7.3. Code assignments for levels 0, 0, 1, 2, 3, 4, ..., $h-1$ and four consecutive operations: 1. DELETE($h-1$), 2. INSERT(0), 3. DELETE(0), 4. INSERT($h-1$).

THEOREM 7.4. *Any algorithm A_I satisfying invariant* (7.1) *is* $\Omega(h)$*-competitive.*

PROOF. Consider the sequence of code insertions on levels $0, 0, 1, 2, 3, 4, \ldots, h-1$. For these insertions, there is a unique code assignment satisfying invariant (7.1), see Figure 7.3. Consider now two requests—deletion of the code at level $h-1$ and insertion of a code on level 0. Then A_I has to move every code on level $l \geq 1$ to the right to create space for the code assignment on level 0 and maintain the invariant (7.1). This takes one code assignment and $h-2$ reassignments. Consider as the next requests the deletion of the third code on level zero and an insertion on level $h-1$. Again, to maintain the invariant (7.1), A_I has to move every code on level $l \geq 1$ to the left. This takes again one code assignment and $h-2$ reassignments. An optimal algorithm can handle these four requests with two assignments, since it can assign the third code on level zero in the right subtree, where A_I assigns the code on level $h-1$. Repeating these four requests k times, the total cost of algorithm A_I is then $C_I = h+1+2k(h-1)$, whereas OPT has $C_{OPT} = h+1+2k$. As k goes to infinity, the ratio C_A/C_{OPT} becomes $\Omega(h)$. □

7.2. *Greedy Strategies.* Assume we have a deterministic algorithm A that solves the one-step offline CA problem. This A immediately leads to a greedy online strategy. As an optimal algorithm breaks ties in an unspecified way, the online strategy can vary for different optimal one-step offline algorithms.

THEOREM 7.5. *Any deterministic greedy online strategy, i.e. a strategy that minimizes the number of reassignments for every insertion and deletion, is* $\Omega(h)$ *competitive.*

PROOF. Assume that A is a fixed, greedy online strategy. First we insert $N/2$ codes at level 1. As A is deterministic we can now delete every second level-1 code, and insert $N/2$ level-0 codes. This leads to the situation depicted in Figure 7.4. Then we delete two codes at level $l = 1$ (as A is deterministic it is clear which codes to delete) and immediately assign a code at level $l+1$. As it is optimal (and up to symmetry unique) algorithm A moves two codes as depicted. The optimal strategy arranges the level-1 codes in a way that it does not need any additional reassignments. We proceed in this way along level 1 in the first round, then left to right on level 2 in a second round, and continue toward the root. Altogether we move $N/4$ codes in the first round and we assign $N/2^3$ codes. In general, in every round i we move $N/4$ level-0 codes and assign $N/2^{i+2}$ level-i codes. Altogether the greedy strategy needs $\mathcal{O}(N) + (N/4)\Omega(\log N) = \Omega(N \log N)$

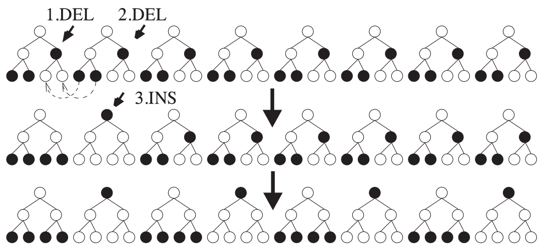


Fig. 7.4. Requests that a greedy strategy cannot handle efficiently.

(re-)assignments, whereas the optimal strategy does not need any reassignments and only $\mathcal{O}(N)$ assignments. □

7.3. *Minimizing the Number of Blocked Codes*. The idea of minimizing the number of blocked codes is mentioned in [23] but not analyzed at all. In every step the algorithm tries to satisfy the invariant:

(7.3) the number of blocked codes in T is minimum.

In Figure 7.5 we see a situation that does not satisfy the invariant (7.3). Moving a code reduces the number of blocked codes by one. We can prove that this approach is equivalent to minimizing the number of gap trees on every level (Lemma 7.7). Recall that a gap tree is a maximal subtree of unblocked codes.

DEFINITION 7.6. The level of the root of a gap tree is called the *level of the gap tree*. The vector $q = (q_0, \ldots, q_h)$, where q_i is the number of gap trees on level i, is called the *gap vector* of the tree T.

See Figure 7.6 for an example of the definition. Invariant (7.3) implies that there is at most one gap tree on every level. Having two gap trees on a level l we can move the sibling tree of one of the gap trees to fill the other gap tree, reducing the number of blocked codes by at least one (concept from Figure 7.5). Also the other direction of this implication holds as it is stated in the following lemma.

LEMMA 7.7. *Let T be a code tree for requests σ. Then T has at most one gap tree on every level if and only if T has a minimum number of blocked codes.*

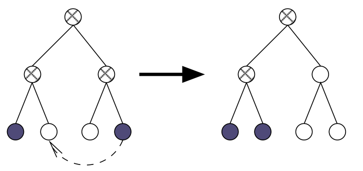


Fig. 7.5. Reassignment of one code reduces the number of blocked codes from three to two.

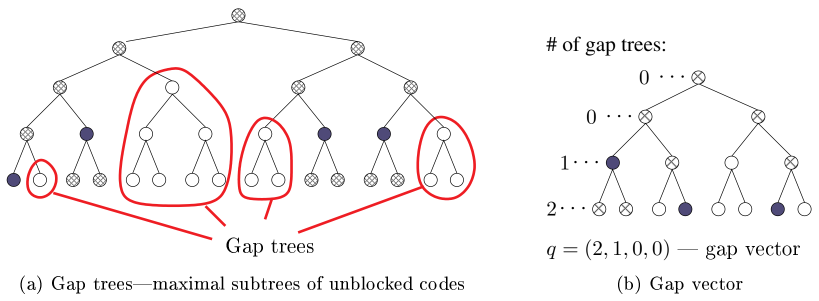

Gap trees
(a) Gap trees—maximal subtrees of unblocked codes
# of gap trees:
0
0
1
2
$q = (2, 1, 0, 0)$ — gap vector
(b) Gap vector

Fig. 7.6. Definition of gap trees and gap vector.

PROOF. We prove only the second implication. Suppose T has at most one gap tree on every level. Since for every tree with two or more gap trees on some level, we can reduce the number of blocked codes by filling the gap trees, the minimum number of blocked codes has to be attained at trees with at most one gap tree at every level. The free bandwidth capacity of T can be expressed as

$$cap = \sum_{i=0}^{h} q_i 2^i.$$

As $q_i \leq 1$, the gap vector is the binary representation of the number *cap* and therefore the gap vector q is unique for every tree serving requests σ with at most one gap tree at every level. The gap vector determines also the number of blocked codes:

$$\#\text{ blocked codes} = (2^{h+1} - 1) - \sum_{i=0}^{h} q_i (2^{i+1} - 1).$$

Thus, every tree for requests σ with at most one gap tree at every level has the same number of blocked codes. □

Now we are ready to define algorithm A_{gap} (Algorithm 7.8). As we will show, on insertions A_{gap} never needs any extra reassignments.

ALGORITHM 7.8 (Algorithm A_{gap}).

1. *Insert*:
   - *Assign the new code into the smallest gap tree where it fits.*
2. *Delete*:
   - *If after the deletion a second gap tree appears on some level, move one of their sibling subtrees into the second gap tree.*
   - *Treat all newly created second gap trees on higher levels recursively.*

LEMMA 7.9. *Algorithm A_{gap} always has a gap tree of sufficient height to assign a code on level l and at every step the number of gap trees at every level is at most one.*

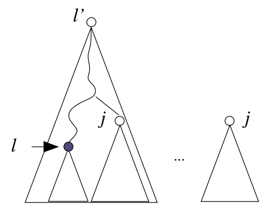


Fig. 7.7. Two gap trees on a lower level than l' violate the minimum chosen height of the gap tree.

PROOF. We know that there is sufficient capacity to serve the request, i.e. $cap \geq 2^l$. We also know that $cap = \sum_i q_i 2^i$. Since $q_i \leq 1$ for all i, there exists a gap tree on level $j \geq l$.

Next, consider an insertion into the smallest gap tree of level l' where the code fits. New gap trees can occur only on levels $j, l \leq j < l'$, and only within the gap tree on level l'. Also, at most one new gap tree can occur on every level. Suppose that after creating a gap tree on level j, we have more than one gap tree on this level. Then, since $j < l'$, we would assign the code into this smaller gap tree, which contradicts our assumption (Figure 7.7). Therefore, after an insertion there is at most one gap tree on every level.

Consider now a deletion of a code. The nodes of the subtree of that code become unblocked, i.e. they belong to some gap tree. At most one new gap tree can occur in the deletion operation (and some gap trees may disappear). Thus, when the newly created gap tree is the second one on the level, we fill the gap trees and then we recursively handle the newly created gap tree on a higher level. In this way the gap trees are moved up. Because we cannot have two gap trees on level $h - 1$, we end up with a tree with at most one gap tree on each level. □

The result shows that the algorithm is optimal for insertions only. It does not need any extra code movements, contrary to the compact representation algorithm. Similarly to the compact representation algorithm, this algorithm is $\Omega(\log N)$-competitive.

THEOREM 7.10. *Algorithm A_{gap} is $\Omega(h)$-competitive.*

PROOF. The proof is basically identical with the proof of Theorem 7.5. □

Algorithm A_{gap} has even a very bad worst case number of code movements. Consider the four subtrees on level $h-2$, where the first one has $N/4$ leaf codes inserted, its sibling has a code on level $h-2$ inserted and the third subtree has again $N/4$ leaf codes inserted (Figure 7.8). After deletion of the code on level $h - 2$, A_{gap} is forced to move $N/4$ codes. This is much worse than the worst case for the compact representation algorithm. Nevertheless, it would be interesting to investigate the best possible upper bound that can be proved for the competitive ratio of A_{gap}.

7.4. *Resource Augmented Online Algorithm.* In this section we present the online strategy 2-*gap* and study it by a resource-augmented competitive analysis. This type of analysis was introduced by Kalyanasundaram and Pruhs [18]. In a resource-augmented

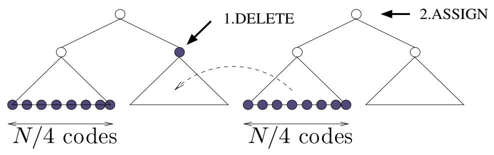


Fig. 7.8. Worst case number of movements for algorithm A_{gap}.

competitive analysis one compares the value of the solution found by the online algorithm, when it is provided with more resources, to the value of the optimal offline adversary using the original resources. In the case of the OVSF online code assignment problem the resource is the total available bandwidth. The strategy 2-*gap* uses a tree T' of bandwidth $2b$ to accommodate codes whose total bandwidth is b. By the nature of the code assignment we cannot add a smaller amount of additional resource. 2-*gap* uses only an amortized constant number of reassignments per insertion or deletion.

Algorithm 2-*gap* is similar to the compact representation algorithm of Section 7.1 (insisting on the ordering of codes according to their level, Invariant (7.1)), only that it allows for up to two gaps at each level ℓ (instead of only one for aligning), to the right of the assigned codes on ℓ. The algorithm for inserting a code at level ℓ is to place it at the leftmost gap of ℓ. If no such gap exists, we reassign the leftmost code of the next higher level $\ell+1$, creating two gaps (one of them is filled immediately by the new code) at ℓ. We repeat this procedure toward the root. We reject an insertion if the nominal bandwidth b is exceeded. For deleting a code c on level ℓ we move the rightmost code on level ℓ into the position c, keeping all codes at level ℓ to the left of the gaps of ℓ. If this results in three consecutive gaps, we reassign the rightmost code of level $\ell+1$, in effect replacing two gaps of ℓ by one of $\ell+1$. Again we proceed toward the root.

More precisely, we keep for every level a range of codes (and gaps) that are assigned to this level. In every range at most two gaps are allowed. We denote by r_ℓ the number of assigned codes in the range of level ℓ and by $q_\ell \in \{0, 1, 2\}$ the number of gap nodes in the same range. The alignment condition that all ranges up to the range of level $h-3$ have to satisfy is the following:

$$\forall k \in \{0, \ldots, h-3\}, \qquad \sum_{\ell=0}^{k} 2^{\ell}(r_\ell + q_\ell) \in 2^{k+1} \cdot \mathbb{N}.$$

This notion of a range is in particular important for levels without codes. The levels close to the root are handled differently, to avoid excessive space usage. The root-code of T' has bandwidth $2b$, it is never used. The bandwidth b code of level h can only be used if no other code is used, there is no interaction with other codes. The $b/2$ codes of level $h-1$ are kept compactly to the right. In general there is some unused bandwidth between the $b/4$ and the $b/2$ codes, which is not considered a gap. This free space can be used to assign codes of bandwidth $b/4$ if by doing so the nominal bandwidth b is not exceeded. Figure 7.9 shows an example code assignment that follows the rules of the 2-*gap* strategy for keeping the codes and gaps of a certain level in ranges. Whenever a code is assigned on a level ℓ the number of codes in the range of level ℓ increases by one and the number of gaps in the range decreases by one. In the case of a code deletion on

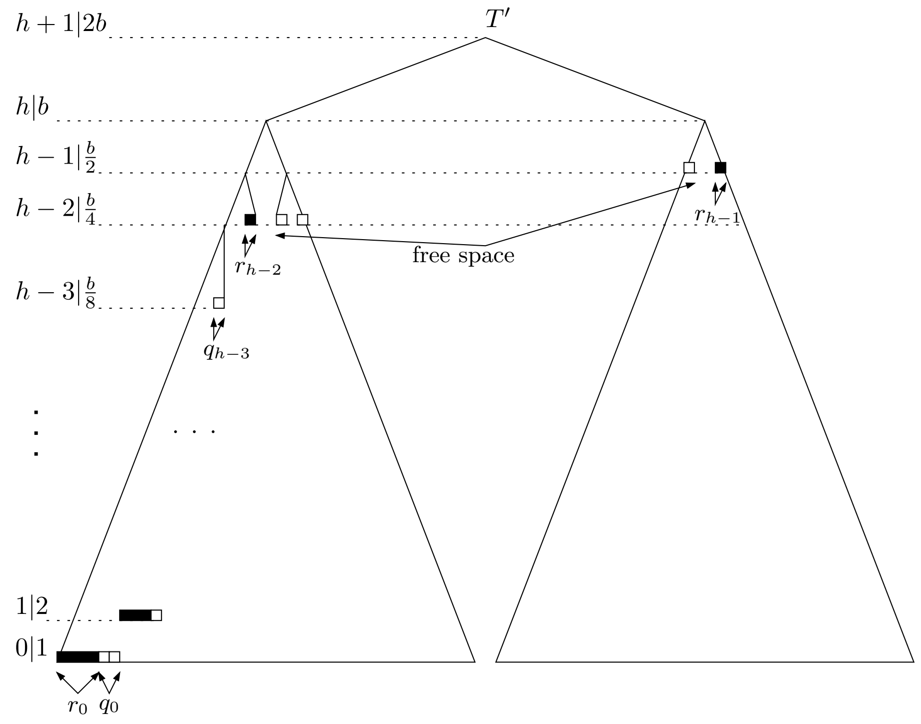


Fig. 7.9. Example code assignment following the rules of the 2-*gap* strategy (the labels on the left side $\ell|j$ show the level ℓ followed by the bandwidth j of the codes assigned on this level).

level ℓ the number of codes in the range of level ℓ decreases by one and the number of gaps increases by one. The critical cases that change the range alignments are:

- A code has to be assigned on level $\ell \leq h-3$ but there is no gap on this level, $q_\ell = 0$. In such cases we assign the code to the right of the range of level ℓ. Hence, the number of codes in the range of level ℓ is increased by one, i.e. $r_\ell \leftarrow r_\ell + 1$, and the number of gaps is temporarily decreased to $q_\ell = -1$. To bring the number of gaps back to the allowed values, the algorithm proceeds as follows. Let $\ell + i$ be the closest higher level (but below level $h-2$) that has at least one assigned code or gap, or let $\ell + i = h-2$ if no such level exists. Let v be the leftmost node on level $\ell + i$ in the range of $\ell + i$. If v is occupied by an assigned code, that code is reassigned to the right of the rightmost code on level $\ell + i$. In the subtree rooted at v this creates, from left to right, two gaps on level ℓ and one gap on every level $\ell' \in \{\ell+1, \ldots, \ell+i-1\}$. The number of gaps on levels ℓ to $\ell+i$ changes according to $q_{\ell+i} \leftarrow q_{\ell+i} - 1$, $q_\ell \leftarrow q_\ell + 2$ and $q'_\ell \leftarrow 1$ for $\ell' \in \{\ell+1, \ldots, \ell+i-1\}$. This results in $q_\ell = 1$. If $q_{\ell+i} = -1$, this is treated like a critical insertion on level $\ell+i$ and the procedure propagates to level $\ell+i$ (unless $\ell + i = h-2$).
- A code is deleted from level $\ell \leq h-3$ whose range already has two gaps, $q_\ell = 2$. After the deletion (and a possible reassignment on level ℓ to keep the gaps placed compactly at the right end of the range) the number of codes on level ℓ becomes

$r_\ell \leftarrow r_\ell - 1$ and the number of gaps in the range of level ℓ temporarily increases to three. To bring the number of gaps back to the allowed values the rightmost two gaps on level ℓ are merged into a level $\ell + 1$ gap, i.e. $q_{\ell+1} \leftarrow q_{\ell+1} + 1$ and $q_\ell \leftarrow q_\ell - 2$. If the range of level $\ell + 1$ has some assigned codes then the rightmost code is moved to fill in the newly created gap. We consider recursively a critical deletion on level $\ell + 1$ in case the number of gaps on level $\ell + 1$ exceeds two after the gap merging on level ℓ (except if $\ell + 1 = h - 2$, in which case nothing more needs to be done).

To compensate the reassignments on levels $\ell \leq h - 3$ (codes of bandwidth $\leq b/8$) we define a potential function computed as the sum of the number of levels without gaps, and the number of levels having two gaps. With this potential function it is clear that it is sufficient to charge two (re-)assignments to every insertion or deletion, one for placing the code (filling the gap), and one for the potential function or for moving a $b/4$-bandwidth code. The initial configuration is the empty tree, where the leaf-level has two gaps, and all other levels have precisely one gap (only the close-to-root levels are as described above). Note that the potential of the initial configuration is equal to 1 and that the potential always remains non-negative.

It remains to show that our algorithm manages to host codes as long as the total bandwidth used does not exceed b. The total bandwidth used in T' is the sum of the bandwidth wasted on gaps, which is at most $2(b/8 + b/16 + \cdots) \leq b/2$, and the nominal bandwidth b that can be assigned. This adds up to $3b/2$ and is less than the bandwidth $2b$ of the tree T'.

THEOREM 7.11. *Let σ be a sequence of m code insertions and deletions for a code-tree of height h, such that at no time is the bandwidth exceeded. Then the above online strategy uses a code-tree of height $h + 1$ and performs at most $2m + 1$ code assignments and reassignments.*

COROLLARY 7.12. *The above strategy is four-ompetitive for resource augmentation by a factor of two.*

PROOF. Any sequence of m operations contains at least $m/2$ insert operations. Hence the optimal offline solution needs at least $m/2$ assignments, and the above resource augmented online-algorithm uses at most $2m + 1$ (re-)assignments, leading to a competitive ratio of 4. □

This approach might prove to be useful in practice, particularly if the code insertions only use half the available bandwidth.

8. Fixed Parameter Tractability of the Problem.

In this section we consider the fixed parameter tractability of the parameterized one-step offline CA problem. Parameterized problems are described by languages $L \subseteq \Sigma^* \times \mathbb{N}$. If $(x, k) \in L$, we refer to k as the parameter. The concept of fixed parameter tractability is described for example in [11].

DEFINITION 8.1 [11]. We say that L is *uniformly fixed-parameter tractable* if there is an algorithm A, a constant c and a function $f\colon \mathbb{N} \to \mathbb{N}$ such that

1. the running time of $A(x, k)$ is at most $f(k)|x|^c$ and
2. $(x, k) \in L$ if and only if A accepts (x, k).

We assume that our problem is given by a pair (x, k), where x encodes the code insertion on level l and the current code assignment and k is the parameter. We assume the encoding of the code assignment in the zero-one vector form $x_1, \ldots, x_{2^{h+1}-1}$ saying for every node of the tree whether there is an assigned code. For the purpose of this section denote by n the size of the input, i.e. $n := |x| = 2^{h+1} - 1$.

We consider various variants of parameters for the problem. The most natural ones are the number of moved codes m or the level l of the code insertion. To show the fixed-parameter tractability, we reuse the ideas of the exact algorithm using dynamic programming, where we store at every node a table of all possible signatures.

We first show that the problem is fixed-parameter tractable, if the parameters are both m and l, i.e. we show an algorithm solving the problem in time $\mathcal{O}(f(m, l)p(n))$ for some polynomial $p(n)$.

Having a code insertion into the code tree for level l, we know that we only move codes from lower levels than l. Hence, when building the tables at nodes, we consider only those signatures that differ on levels $0, \ldots, l-1$ from the signature of the current subtree. From the assumption that we move at most m codes, we have that on each of these levels, the considered signature can differ by at most m. Hence, the number of considered signatures in every node is at most $(2m+1)^l$. To compute all the tables, we need to combine all the tables from the children nodes, i.e. we have to consider $(2m+1)^{2l}$ pairs for every node. From this we get the running time $\mathcal{O}(2^h(2m+1)^{2l})$, which is certainly of the form $f(m, l)p(n)$.

For the case, where we have only l as the parameter, we immediately get that we move from every subtree T_v at most 2^l codes, hence we bound the number of codes moved in every subtree by a parameter (we note that we did not bound the overall number of moved codes) $m = 2^l$.

Consider now the case where only m is the parameter. Since we move at most m codes within the tree, we know that at most m codes come into the subtree and at most m go away from the subtree. Hence, assigning for each such possibility a level out of $0, \ldots, l$, we get an upper bound of at most $(l+1)^{2m}$ signatures to be considered at every node on level l. Since $l+1 \leq h$ for $l = 0, \ldots, h-1$ we get at every node at most $h^{2m} = \log n^{2m}$ signatures. From [22] we can use the inequality $(\log n)^m \leq (3m \log m)^m + n$ to express the size of each table in the form $g(m) + n$. To compute the table for every node, we need time $n(g(m) + n)^2$ which is certainly of the form $f(m)p(n)$.

We can summarize the results of this section in the following theorem.

THEOREM 8.2. *The one-step offline CA problem is fixed-parameter tractable for the following parameters*:

- *level l of the code insertion and*
- *the number m of moved codes.*

9. Conclusions and Future Work. In this paper we bring an algorithmically interesting problem from the mobile telecommunications field closer to the theoretical computer science community. We are the first to analyze the computational complexity of the OVSF code assignment problem. We point out that the algorithm in [21], believed to have solved the one-step offline CA problem, is erroneous and we prove that, for a natural encoding of the input, the problem is *NP*-complete. We present an exact algorithm for the one-step offline CA problem that has running time $n^{\mathcal{O}(h)}$. We also prove that the simplest greedy algorithm for the one-step offline version is an h-approximation algorithm. Next we introduce and analyze the more realistic online version of the problem. For insertions and deletions the online strategy that uses the compact representation is $\mathcal{O}(h)$ competitive. We also show that a slight modification of the compact representation algorithm that uses only twice the available bandwidth is 4-competitive.

Future research on CA could concentrate on the following open problems:

- Is there a constant approximation algorithm for the one-step offline CA problem?
- Can the gap between the lower bound of 1.5 and the upper bound of $\mathcal{O}(h)$ for the competitive ratio of the online CA be closed?
- Is there an instance where the optimal general offline algorithm has to reassign more than an amortized constant number of codes per insertion or deletion?

References

[1] J. Adamek. *Foundation of Coding*. Wiley, Chichester, 1991.

[2] F. Adashi, M. Sawahashi, and K. Okawa. Tree structured generation of orthogonal spreading codes with different lengths for forward link of DS-CDMA mobile radio. *Electronics Letters*, 33(1):27–28, January 1997.

[3] R. Assarut, M. G. Husada, U. Yamamoto, and Y. Onozato. Data rate improvement with dynamic reassignment of spreading codes for DS-CDMA. *Computer Communications*, 25(17):1575–1583, 2002.

[4] R. Assarut, K. Kawanishi, R. Deshpande, U. Yamamoto, and Y. Onozato. Performance evaluation of orthogonal variable-spreading-factor code assignment schemes in W-CDMA. In *IEEE International Conference on Communications* (*ICC*), pages 3050–3054, 2002.

[5] A. Borodin and R. El-Yaniv. *Online Computation and Competitive Analysis*. Cambridge University Press, Cambridge, 1998.

[6] H. Çam. Nonblocking OVSF codes and enhancing network capacity for 3G wireless and beyond systems. *International Conference on Third Generation Wireless and Beyond*, pages 148–153, May 2002.

[7] J.-C. Chen and W.-S. E. Chen. Implementation of an efficient channelization code assignment algorithm in 3G WCDMA. In *Proceedings of National Computer Symposium*, pages E237–E244, 2001.

[8] W. T. Chen and S. H. Fang. An efficient channelization code assignment approach for W-CDMA. In *IEEE Conference on Wireless LANs and Home Networks*, 2002.

[9] W. T. Chen, Y. P. Wu, and H. C. Hsiao. A novel code assignment scheme for W-CDMA systems. In *Proceedings of the* 54*th IEEE Vehicular Technology Society Conference*, volume 2, pages 1182–1186, 2001.

[10] M. Dell'Amico, F. Maffioli, and M. L. Merani. A tree partitioning dynamic policy for OVSF codes assignment in wideband CDMA. *IEEE Transactions on Wireless Communications*, 3(4):1013–1017, 2004.

[11] R. G. Downey and M. R. Fellows. *Parametrized Complexity*. Monographs in Computer Science. Springer, New York, 1999.

[12] R. Fantacci and S. Nannicini. Multiple access protocol for integration of variable bit rate multimedia traffic in UMTS/IMT-2000 based on wideband CDMA. *IEEE Journal on Selected Areas in Communications*, 18(8):1441–1454, August 2000.

[13] A. Fiat and G. J. Woeginger. *Online Algorithms*. Volume 1442 of Lecture Notes in Computer Science. Springer, Berlin, 1998.

[14] C. E. Fossa, Jr. Dynamic Code Sharing Algorithms for IP Quality of Service in Wideband CDMA 3G Wireless Networks. Ph.D. thesis, Virginia Polytechnic Institute and State University, April 2002.

[15] C. E. Fossa, Jr., and N. J. Davis, IV. Dynamic code assignment improves channel utilization for bursty traffic in 3G wireless networks. In *IEEE International Conference on Communications* (*ICC*), pages 3061–3065, 2002.

[16] M. R. Garey and D. S. Johnson. *Computers and Intractability*. Freeman, San Francisco, CA, 1979.

[17] H. Holma and A. Toskala. *WCDMA for UMTS*. Wiley, New York, 2001.

[18] B. Kalyanasundaram and K. Pruhs. Speed is as powerful as clairvoyance. *Proceedings of the* 36*th IEEE Symposium on Foundations of Computer Science*, pages 214–221, 1995.

[19] A. C. Kam, T. Minn, and K.-Y. Siu. Supporting rate guarantee and fair access for bursty data traffic in W-CDMA. *IEEE Journal on Selected Areas in Communications*, 19(11):2121–2130, November 2001.

[20] J. Laiho, A. Wacker, and T. Novosad. *Radio Network Planning and Optimisation for UMTS*. Wiley, New York, 2002.

[21] 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.

[22] V. Raman, S. Saurabh, and C. R. Subramanian. Faster fixed parameter tractable algorithms for undirected feedback vertex set. In *Proceedings of the* 13*th International Symposium on Algorithms and Computation* (*ISAAC*), pages 241–248. Volume 2518 of Lecture Notes in Computer Science. Springer, Berlin, 2002.

[23] A. N. Rouskas and D. N. Skoutas. OVSF codes assignment and reassignment at the forward link of W-CDMA 3G systems. In *Proceedings of the* 13*th IEEE International Symposium on Personal*, *Indoor and Mobile Radio Communications* (*PIMRC*), volume 5, pages 2404–2408, 2002.