Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Thomas Erlebach, Riko Jacob, Matúš Mihaľák,
Marc Nunkesser, Gábor Szabó, Peter Widmayer

An Algorithmic View on OVSF Code Assignment

TIK-Report
Nr. 173, August 2003

Institut für Technische Informatik und Kommunikationsnetze
Computer Engineering and Networks Laboratory

Thomas Erlebach, Riko Jacob, Matúš Mihaľák, Marc Nunkesser, Gábor Szabó, Peter Widmayer
An Algorithmic View on OVSF Code Assignment
August 2003
Version 1
TIK-Report Nr. 173

Computer Engineering and Networks Laboratory,
Swiss Federal Institute of Technology (ETH) Zurich

Institut für Technische Informatik und Kommunikationsnetze,
Eidgenössische Technische Hochschule Zürich

Gloriastrasse 35, ETH Zentrum, CH-8092 Zürich, Switzerland

An algorithmic view on OVSF code assignment*

Thomas Erlebach† Riko Jacob‡ Matúš Mihaľák† Marc Nunkesser‡
Gábor Szabó‡ Peter Widmayer‡

29th August 2003

Abstract

Orthogonal Variable Spreading Factor (OVSF) codes can be used to share the radio spectrum among several connections of possibly different bandwidth, which is used for example in UMTS. 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 2^{-d} 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, reassign a minimum number of codes to serve a new request. Minn and Siu propose the so-called DCA algorithm to solve the problem optimally. 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 $\Theta(h)$. We also consider 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.

*Research partially supported by EU Thematic Network APPOL II, IST-2001-32007, with funding by the Swiss Federal Office for Education and Science.

†Computer Engineering and Networks Laboratory (TIK), Department of Information Technology and Electrical Engineering, ETH Zürich, {erlebach,mihalak}@tik.ee.ethz.ch

‡Department of Computer Science, ETH Zürich, {jacob,nunkesser,szabog,widmayer}@inf.ethz.ch

1 Introduction

Recently UMTS[1] 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[2] that turns out to be of algorithmic interest, more precisely on its multiple access method DS-CDMA.[3] The purpose of this access method is to make it possible for 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 base-station to the different users. More precisely, we consider spreading by Orthogonal Variable Spreading Factor (OVSF) codes [14, 2], which are used on the downlink and the dedicated channel of the uplink. These codes are derived from a code tree. 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. In particular, it is irrelevant which code on a level a user gets, as long as all codes are mutually orthogonal.

The OVSF code tree is a complete binary tree that reflects the construction of Hadamard matrices: 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 $(a, -a)$. Obviously, all codes on one level are mutually orthogonal. In the DS-CDMA system users request different data rates and get OVSF codes of different levels. (The data rate is inversely proportional to the length of the code.) All codes that are assigned to users are mutually orthogonal if and only if on each path from a leaf to the root there is at most one assigned code. 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.

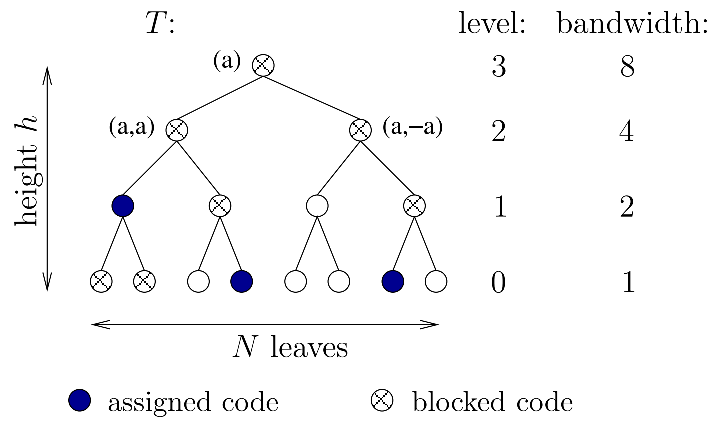


Figure 1.1: A code assignment and blocked codes

[1]Universal Mobile Telecommunications System, for an in-depth coverage of the technical underpinnings we refer the reader to the literature [14, 16].

[2]Wideband Code Division Multiple Access

[3]Direct Sequence Code Division Multiple Access

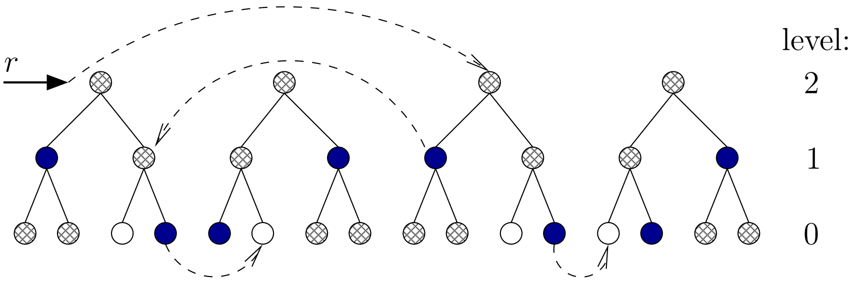


Figure 1.2: A request r for a code and one of the possible reassignments (dashed arcs)

As users connect to and disconnect from a given base station, i.e. request and release codes, the assigned codes in the code tree can get fragmented, so that it can happen that a code request for a higher level cannot be served at all, because lower level codes block *all* codes on the specified level. This problem is known as *code blocking* or *code tree fragmentation* in the literature [16, 17]. One way of solving this problem is to reassign some codes in the tree (more precisely, to assign changed OVSF codes to some users in the cell, without changing the level of the codes). In the example in Figure 1.2 a user requests a code on level two, where all codes are blocked. Still, after reassigning some of the already assigned codes, the code request can be served.

The process of reassigning codes necessarily induces signaling overhead from the base station to the users whose codes change, which should be kept small. Therefore, a natural objective already stated in [17, 18] 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.) The problem has been studied before with focus on simulations. In [17] the problem of reassigning the codes for a single additional request is defined. The Dynamic Code Assignment (DCA) algorithm is introduced and claimed to be optimal. In this paper we prove that the DCA algorithm is not always optimal and analyze natural versions of the underlying code assignment (CA) problem. Our intention is to present a first rigorous analysis of a problem that looks simple at first glance but turns out to be algorithmically challenging after closer inspection. Still, interesting questions remain open.

First, we give a counterexample to the optimality of the DCA algorithm in Section 2. In Section 3 we begin our analysis of code assignment problems with some simple observations. Then we prove the original problem stated by Minn and Siu [17] to be *NP*-complete for a natural input encoding in Section 4. In Section 5 we give a dynamic programming algorithm that solves the problem optimally and is efficient for small instance sizes. In Section 6 we show that a natural greedy algorithm already mentioned in [17] achieves approximation ratio h for one step. We tackle the online problem[4] in Section 7, which is a more natural version of the problem, because we are interested in minimizing the signaling overhead over all operations rather than in every step. We present a $\Theta(h)$-competitive algorithm

[4]We use standard terminology from the field of online algorithms, see e.g. [10].

and show that the greedy strategy that minimizes the number of reassignments in every step is not better than $\Omega(h)$-competitive. We also give an online algorithm with constant competitive ratio that uses resource augmentation, i.e. we give it one more level than the adversary. In Section 8 we show that a very natural class of algorithms can be forced into arbitrary configurations. This finding gives more substance to both the *NP*-completeness proof and the analysis of possible online algorithms. We draw our conclusions in Section 9. Appendix A contains additional details and an example for the dynamic programming algorithm of Section 5.

1.1 Problem definition and preliminaries

We are concerned with assigning codes of an (OVSF) code tree $T = (V, E)$ to users. As the tree is a complete binary tree, it is completely specified by its height h. All users who are 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 tells us how many users request a code on level i (with bandwidth 2^i, where we assume without loss of generality that the leaf codes have bandwidth 1). We count the levels from leaves to root. The level of a node, a code, or a code request c is denoted by $l(c)$. The subtree of T rooted at a node v is denoted by T_v. The requests have to be mapped to positions (nodes) in the tree, such that:

1. For all levels $i \in \{0 \dots h\}$ there are exactly r_i codes on level i.

2. On every path p_j from a leaf j to the root there must be at most one code.

We call every set of positions in T that fulfills these properties a *code assignment* $F \subset V$. For ease of presentation we call F sometimes also a set of *codes.* A set $S \subseteq V$ that fulfills only condition two is called *independent*, and its elements are called *independent positions.* Changing a position in F is called *moving* a code. When we talk about a code tree, we usually mean the tree together with a code assignment F. If an additional user requests a code we call this a *code request* (on a given level), if some user disconnects we call this a *deletion* (in a given position). A code request is also called code *insertion.* The request is dropped if it cannot be served because its acceptance would exceed the total bandwidth. By N we denote the number of leaves of T and by n the number of assigned codes $|F|$. After a request on level l_t at time t any correct CA algorithm must change the code assignment F_t into a code assignment F_{t+1} for the new request vector $r' = (r_0, \dots, r_{l_t} + 1, \dots, r_h)$. We call $|F_{t+1} \setminus F_t|$ the number of *reassignments.* This implies that in the case of an insertion, we also consider the one new assignment as a reassignment and count it as such. We do not count deletions because every code can be deleted only once. This is important, because we take the number of reassignments as the cost function.

We state the original CA problem studied by Minn and Siu [17] 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 $r' = (r_0, \dots, r_l + 1, \dots, r_h)$ with minimum number of reassignments.

general offline CA Given a sequence S of code requests and deletions of length m, find a sequence of code assignments so that the total number of reassignments is minimum, assuming the initial code tree is empty.

online CA This is the same problem as the general offline CA, except that the future requests of S are not known in advance.

insertion-only online CA Here S consists of insertions only.

It is instructive to look at a natural ILP formulation of the one-step offline CA. We introduce $\{0,1\}$-decision variables x_v that decide whether there is a code at position $v \in V$ in the tree or not.

$$\begin{array}{lrcll} \max & \sum_{v \in F} & x_v & & \\ \text{s.t.} & \sum_{v \in l_i} x_v & = & r'_i & \forall \text{ levels } l_i \\ & \sum_{v \in p_j} x_v & \leq & 1 & \forall \text{ paths } p_j \\ & x_v & \in & \{0,1\} & \end{array}$$

In this formulation we are maximizing over the positions that stay, i.e. over x_v where v is in the initial assignment F, which is equivalent to minimizing the number of moved codes.

1.2 Related work

It was a paper by Minn and Siu [17] that originally drew our attention to this problem. There the one-step CA version is defined together with an algorithm that is claimed to solve it optimally. As we show in Section 2 this is not true, the argument contains errors. Many of the follow-up papers such as [4, 15, 18, 5, 11, 12] acknowledge the original problem to be solved by Minn and Siu and study some other aspects of it. Assarut et al. [4] do a performance evaluation of Minn and Siu's DCA algorithm, and compare it to 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 [18, 5]. A different direction is to use a heuristic approach that solves the problem for small problem instances [5]. Kam, Minn and Siu [15] address the problem in the context of bursty traffic and different QoS.[5] They come up with a notion of "fairness" and also propose to use multiplexing. Priority based schemes for different QoS classes can be found in [7], similar in perspective are [12, 11].

Fantacci and Nannicini [9] are among the first to express the problem in its online version, although they have quite a different focus. They present a scheme that is similar to the compact-representation scheme in Section 7, without focusing on the number of reassignments. Rouskas and Skoutas [18] 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 [6] propose a best-fit least-recently used approach, also without analysis.

[5]Quality of Service

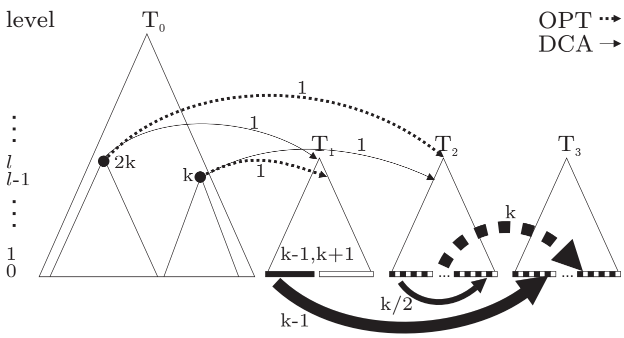


Figure 2.1: Example for the proof of Theorem 2.1

2 Non-optimality of greedy algorithms

Here we look at possible greedy algorithms for the one-step offline CA. A straight-forward greedy approach is to select for a code request a subtree with minimum cost that is not blocked by a code above the request 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 in a minimum cost subtree, takes out the codes in that subtree and proceeds recursively. The DCA algorithm in [17] 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 2.1. *Any top-down greedy algorithm A_{tdg} with a cost function independent of the sequence of codes that have to be reassigned is not optimal.*

As all proposed cost functions in [17] are independent of the sequence of codes to be reassigned, this theorem implies the non-optimality of the DCA algorithm.

Proof. Our construction considers the subtrees in Figure 2.1 and the assignment of a new code in the root of the tree T_0. The tree T_0 has a code with bandwidth $2k$ on level l and depending on the cost function has or does not have 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 choose to put the initial code at the root of T_0. We are left with two cases:

case 1 The cost function evaluates T_2 or 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 $2k$ into the root of the subtree T_2 or T_3, which causes one more reassignment than assigning it into the root of T_1.

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 $2k$ into the root

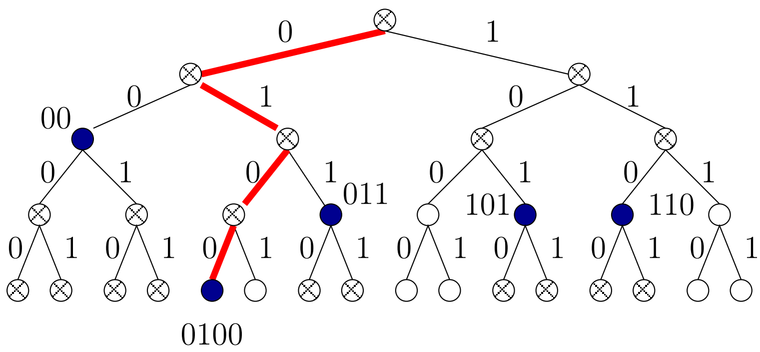


Figure 3.1: Correspondence of code assignments in tree of height 4 with codes on levels {0,1,1,1,2} and prefix free codes of lengths {4,3,3,3,2}

of T_1 and the code with bandwidth k into the tree T_2 or T_3, see solid lines in Figure 2.1. The number of reassigned codes is $3k/2+2$. However, the minimum number of reassignments is $k+3$, achieved when the code with bandwidth k is moved in the empty part of T_1 and the code with bandwidth $2k$ is moved into the root of T_2 or T_3, see dashed lines in Figure 2.1.

□

3 Observations about the code assignment problem

3.1 Feasibility of code assignment

Given a tree T with n codes at levels $l_1, \ldots, l_n$ and a code request for level l_{n+1}, can we serve the request? Can we reassign the codes such that we can assign a code on level l_{n+1}? Such a code reassignment for desired levels $\{l_1, \ldots, l_{n+1}\}$ exists if and only if there exists a binary prefix free code set of given lengths $\{h-l_1, \ldots, h-l_{n+1}\}$. 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 $\{h-l_1, \ldots, h-l_{n+1}\}$ we can clearly assign codes on levels l_i — just follow the paths described by the code words (see Figure 3.1).

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

Theorem 3.1. *A binary prefix code of code lengths $a_1, \ldots, a_m$ exists if and only if*

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

Proof. E.g., in [1]. □

Corollary 3.2. *A code assignment for desired levels $l_1, \ldots, l_m$ into the tree T of height h with N leaves exists if and only if*

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

Proof. According to Theorem 3.1, a binary prefix code exists if and only if $\sum_{i=1}^{m} 2^{-a_i} \leq 1$. Multiplying with 2^h we get $\sum_{i=1}^{m} 2^{h-a_i} \leq 2^h$. Setting $l_i = h - a_i$, i.e., $a_i = h - l_i$, the statement follows. □

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

3.2 Irrelevance of higher level codes

In the one-step offline CA we are looking for a subtree T_v to assign a code for the request c for a code assignment on level $l(c) = l(v)$. In this section we show according to [17] that an optimal algorithm moves only codes on levels smaller than $l(c)$, i.e., it does not move codes on levels greater or equal to $l(c)$.

Lemma 3.3. *Let c be a request for a code assignment on level $l(c)$ into a code tree T. Then for every code reassignment F' that moves a code of level $l \geq l(c)$ there exists a code reassignment F'' moving fewer codes, i.e., with $|F'' \setminus F| < |F' \setminus F|$.*

Proof. Let $x \in F$ be the highest code that is reassigned and suppose $l(x) \geq l(c)$. Let $y \in F'$ be a position in T to which x is assigned. Denote by S the set of codes from T that are moved into T_x and by Q the set of codes in T_y (these two sets need not to be disjoint). Denote by R the rest of the codes that are moved, i.e., $R = (F' \setminus F) \setminus (S \cup Q)$ (the set R contains also the assigned code for the request c). Denote by $Q' \subseteq Q$ the set of codes in Q that are moved into T_x. Then the number of movements can be expressed as $Cost = |S \setminus Q'| + |Q| + |R|$ (where R includes also the movement of x). To construct F'', we assign codes S into T_y (we can do that because the capacity of T_x and T_y is the same) and thus have to reassign only $Q \setminus Q'$ from Q (the codes in Q' are already involved in the reassignments S). The rest of the codes can be reassigned as in F', but with $|R| - 1$ movements since we do not have to move x. We get a new code assignment F'' with cost $Cost' = |R| - 1 + |S \setminus Q'| + |Q'| + |Q \setminus Q'| = Cost - 1$, from which the lemma follows. □

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 asks if a new code request can be handled with cost less or equal to a number $c_{\max}$, which is also part of the input. First of all, we note that the decision variant is in *NP*, because we can guess an optimal assignment and verify in

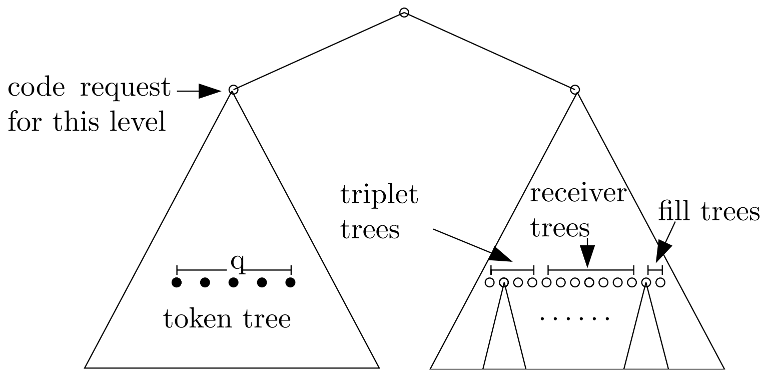


Figure 4.1: Sketch of the construction

polynomial time if it is feasible and if its cost is less 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 (cf. [13]):

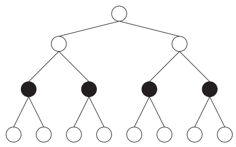
Problem 4.1. (3DM) Given a set $M \subseteq W \times X \times Y$, where W, X and Y are disjoint sets having the same number q of elements, does M contain a matching, i.e., a subset $M' \subseteq M$ such that $|M'| = q$ and no two elements of M' agree in any coordinate?

Let us index the elements of the ground sets 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 see 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.

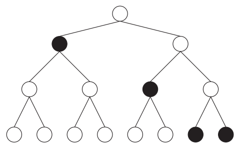
Figure 4.1 shows an outline of the construction that we use for the transformation. An input to 3DM is transformed into an initial feasible assignment that consists of a token tree on the left side and different smaller trees on the right. The construction is set up in such a way that the code request forces the q codes on the left side to move to the right side. Then these codes must be assigned to the roots of triplet trees. The choice of the q triplet trees reflects the choice of the corresponding triplets for a matching. All codes in the chosen triplet trees find a new place without any additional reassignment if and only if these triplets really represent a 3D matching.

Let us now delve into the details of the construction. The token tree consists of q codes positioned arbitrarily on level l_{start} with sufficient depth, e.g. $\lceil \log(|M| + 21q^2 + q) \rceil + 1$. 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).

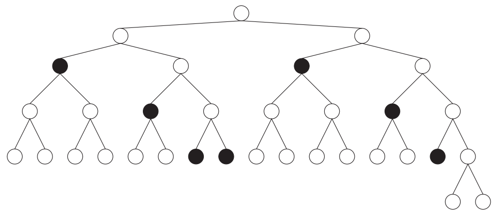
Figures 4.2 (c) and (d) show how layers are stacked using *sibling trees.* We have chosen the zero- and one-trees such that both have the same number of codes and occupy the same bandwidth (but are still different).



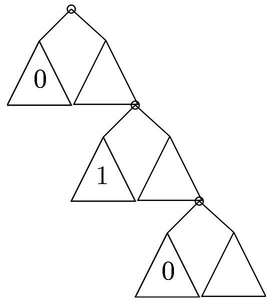
(a) The zero-tree



(b) The one-tree



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



(d) Stacking layers

Figure 4.2: Encoding of zero and one

The receiver trees are meant to receive all codes in the 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 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 these 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. To us, two encodings seem natural:

- 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 hope for 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 ϕ the transformation described above. Then $M \in 3DM$ if and only if $\phi(M)$ can be done with $\alpha = 21q^2 + 2q + 1$ reassignments.*

Proof. Assume there is a matching $M' \subset M$. Now consider the reassignment that assigns the code request 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 $\phi(M)$ with at most α reassignments. Clearly, the initial code request 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. But 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. But by construction the extra zero-tree cannot be assigned to the remaining receiver tree 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 Section 8 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 request level.*

5 Exact $n^{\mathcal{O}(h)}$ dynamic programming algorithm

In this section we describe an exact algorithm for the one-step offline CA problem using dynamic programming, and at the end of the section we give the asymptotic analysis for the storage space required and the running time. We define the *signature* for a subtree at level k as a $k + 1$ dimensional vector. The signature $V_k = [a_k, a_{k-1}, \ldots, a_1, a_0]^T$ tells us that on level i there are a_i assigned codes. To have a feasible signature for a tree T_k, the

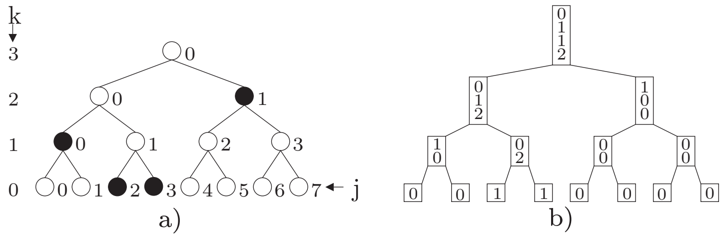


Figure 5.1: a) OVSF code tree. b) Tree configuration using signatures.

vectors must satisfy the capacity constraint $\sum_{i=0}^{k} a_i \cdot 2^i \leq 2^k$. One way to further reduce the number of signatures is by restricting the feasible signatures to those that are relevant for the current input. Let $V_h = [v_h, v_{h-1}, \ldots, v_l, \ldots, v_1, v_0]^T$ be the original tree signature and l the level of the new code request. The root node of the final tree will have the signature $V'_h = [v'_h, v'_{h-1}, \ldots, v'_l, \ldots, v'_1, v'_0]^T = [v_h, v_{h-1}, \ldots, v_l + 1, \ldots, v_1, v_0]^T$. For every node on level k we need only those feasible signatures that satisfy the additional constraint of the input instance $a_i \leq v'_i, \forall i \in \{0 \ldots k\}$.

We define the *tree configuration* as the representation of the OVSF code tree given by the corresponding subtree signatures of the nodes in the tree. Figure 5.1 shows an example tree for the tree configuration. The tree configuration is equivalent to a code assignment (see Section 1.1) for the request vector given by the signature.

The algorithm constructs one 2-dimensional table M_k for each level k starting with the leaf level. The tables have as rows all relevant tree signatures for that level and one column for every node of this level. The table for the root level h has only one entry for the final tree signature V'_h. We use the number of assigned codes that are not moved by the algorithm as a profit function that we want to maximize. Each cell c of a table stores the following values: the *maximum profit* $c.P$, and a *left* and *right* pointer ($c.L$ and $c.R$) to the table entries from one level below that achieve the maximum profit. The cells of table M_k are computed by using the cells in table M_{k-1}. We combine two columns from table M_{k-1} that represent sibling nodes at level $k-1$. We enumerate all possible signature pairings from these columns and update the cell of the corresponding parent signature (the sum of the left and right subtree signature augmented with 0 at position k) in table M_k. The update is done only if the sum of the profits from the left and right subtrees is bigger than the current profit stored in the cell. The left and right pointers are also updated and are used at the end of the algorithm to reconstruct the final tree configuration with the new code assigned. The signature $[1, 0, \ldots, 0, 0]^T$ for a node on level k cannot be constructed using the signatures from table M_{k-1}. In this case the left and right subtrees have the signature $[0, 0, \ldots, 0, 0]^T$ and the maximum profit is 1 if originally there was a code assigned at this node and 0 otherwise. The number of codes that were reassigned by the algorithm is the difference between the number of codes in the tree and the maximum profit for the root node signature. The pseudo-code of the algorithm together with a concrete example is shown in Appendix A.

The algorithm finds a feasible configuration of the OVSF tree, including the new code, so that a maximum number of already assigned codes do not move. For a node j at level k and a possible signature S_k, the maximum number of codes that do not move is computed using the following recursion:

$$M_k[S_k, j].P = \begin{cases} 1, & \text{if } S_k = [1, 0, \ldots, 0, 0]^T \text{ and originally a code was assigned to node } j \\ 0, & \text{if } S_k = [1, 0, \ldots, 0, 0]^T \text{ and originally node } j \text{ was not assigned} \\ \max_{S_{left}+S_{right}=S_k}(M_{k-1}[S_{left}, lj].P + M_{k-1}[S_{right}, rj].P), & \text{otherwise} \end{cases}$$

Here, lj and rj refer to the left and right child of j, respectively. Note that the pairings of subtree signatures S_{left} and S_{right} must be limited to those subtree signatures from level $k-1$ that when summed up and augmented with 0 at position k give the signature S_k. The two problems, maximizing the number of codes that do not move and minimizing the number of reassigned codes, are equivalent. Thus, the correctness of the recursive formula above proves also the correctness of the algorithm for solving the original problem.

For the analysis of the algorithm we need to determine the number of relevant signatures per table and the time to compute the values stored in the cells. For the table M_k the number of columns is 2^{h-k} and the number of rows is the number of relevant tree signatures for level k. Let us denote this number by $\mathcal{C}_k$. We can bound $\mathcal{C}_k$ by $\prod_{i=0}^{k}(v'_i + 1)$. Let $n = \|V'_h\| = \sum_{i=0}^{h} v'_i$ be the total number of assigned codes. Using the arithmetic-geometric mean inequality, it follows that $\mathcal{C}_{h-1} \leq (1 + \frac{n}{h})^h$. Let $\mathcal{S}_h = \sum_{k=0}^{h-1} 2^{h-k} \cdot \mathcal{C}_k$ denote the number of entries in the tables of the algorithm for levels $0, \ldots, h-1$. Using the upper bound for $\mathcal{C}_{h-1}$ and that $\mathcal{C}_i \leq \mathcal{C}_{h-1}$, $\forall i \in \{0 \ldots h\}$, we have $S_h \leq \sum_{k=0}^{h-1} 2^{h-k} \cdot \mathcal{C}_{h-1} \leq 2^h \cdot (1 + \frac{n}{h})^h \cdot \sum_{k=0}^{h-1} \frac{1}{2^k} \leq 2^{h+1} \cdot (1 + \frac{n}{h})^h$. Therefore, the storage space required by the algorithm is bounded by $\mathcal{O}(2^h \cdot (1 + \frac{n}{h})^h)$. An upper bound on the running-time is given in the following theorem.

Theorem 5.1. *The dynamic programming algorithm presented above has asymptotic running time* $n^{\mathcal{O}(h)}$.

Proof. The running time of the algorithm is proportional to $\mathcal{R}_h = 2^{h+1} + \sum_{k=1}^{h} 2^{h-k} \cdot (\mathcal{C}_{k-1})^2 \cdot h$, taking into account the time for computing the entries for table M_0 (first component) and the time for computing the entries for tables $M_k, k \in \{1 \ldots h\}$ (second component). Thus, the running time is $\mathcal{O}(h \cdot 2^h \cdot (1 + \frac{n}{h})^{2h}) = n^{\mathcal{O}(h)}$. □

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 assignment request 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[6]. 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 the greedy algorithm 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 request c, its level is denoted by $l(c)$.

Algorithm 6.1. *Greedy algorithm A_{greedy}:*

```
C_0 ← {c_0}; T^0 ← T
t ← 0
WHILE C_t ≠ ∅ DO
   c_t ← element with highest level in C_t
   g ← the root of a subtree T_g^t 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} ← (T^t \ Γ(T_g^t)) ∪ {g}
   C_{t+1} ← (C_t ∪ Γ(T_g^t)) \ {c_t}
   t ← t + 1
END WHILE
```

In [17] 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 the greedy algorithm 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, the greedy algorithm 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 4 (re-)assignments. Obviously, for this example the greedy algorithm 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} to 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* (cf. 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 to assign the codes in C_t into positions inside the opt-trees. This possibility can be expressed by a code mapping $\phi_t : C_t \to V(\mathcal{T}_{opt})$. The key-property is now that in every step of the algorithm there is the theoretical choice to complete the current assignment using the code mapping ϕ and the opt-arcs as follows: Use ϕ to assign the codes in C_t into

[6]From Lemma 3.3 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.

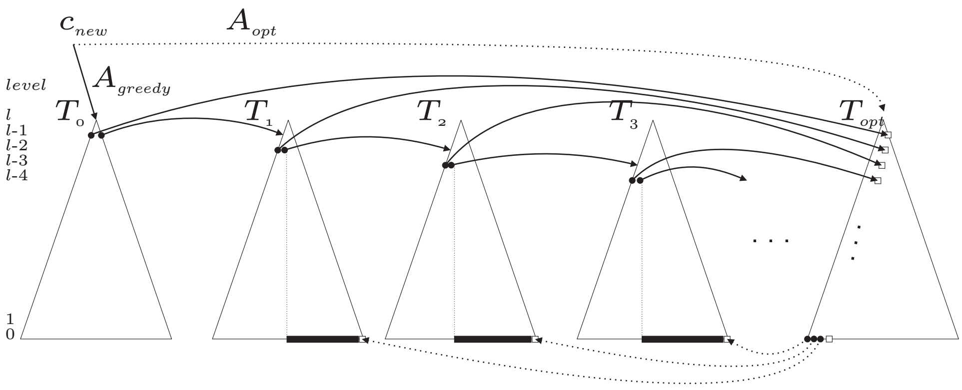


Figure 6.1: Example for the lower bound for A_{greedy}

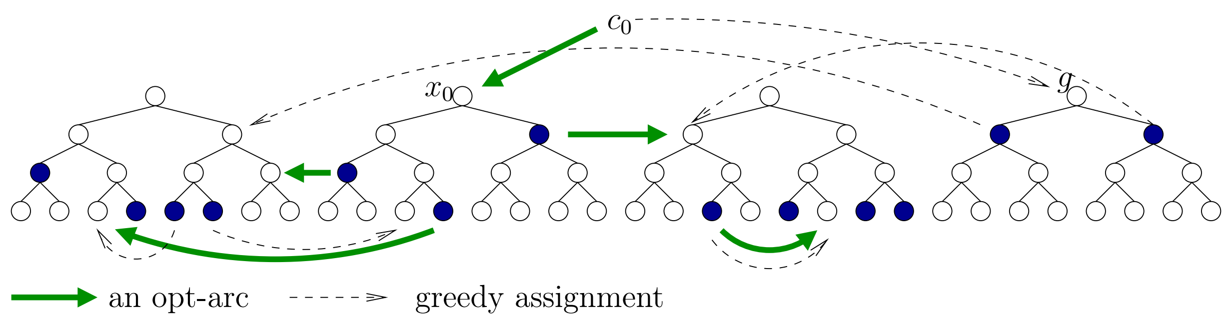


Figure 6.2: A_{opt} moves codes to assign a new code c_0 using opt-arcs. The opt-trees are subtrees to the root of which A_{opt} moves codes. Here, the cost of the optimal solution is 5. The greedy algorithm has cost 6.

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 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 request 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 $\forall v \in M_{\alpha_t} : \ l(v) = l(\alpha(v))$ and $\alpha_t(M_{\alpha_t}) \cup (F^t \setminus M_{\alpha_t})$ is a code assignment.

The set $\alpha_t(M_{\alpha_t}) \cup (F^t \setminus M_{\alpha_t})$ represents the resulting code assignment after reassignment of the codes $M_{\alpha_t} \subseteq F^t$ by α_t.

Definition 6.3. Let T^t be a code tree, x, y be positions in T^t and 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

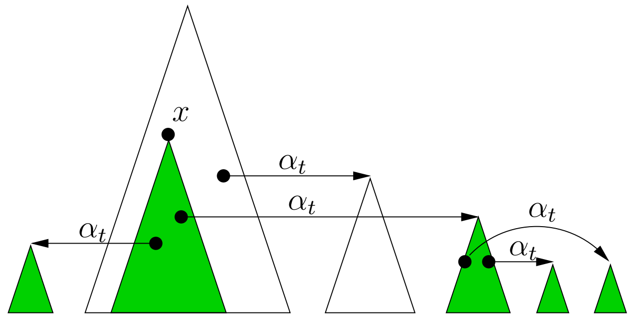


Figure 6.3: The filled subtrees represent all the positions that depend on x.

from a parent to a child and α_t-arcs. By $\text{dep}_t(x)$ 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 $u \in \text{dep}_t(x)$.

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 $\phi_t : C_t \to V(\mathcal{T}_{opt})$ 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 $\phi_t(c)$ and c are the same (i.e. $l(c) = l(\phi_t(c))$.
2. $\forall c \in C_t$ there is no code in T^t at or above the roots of the trees in $\text{dep}_t(\phi_t(c))$.
3. the code movements realized by ϕ_t and α_t (i.e. the set $\phi_t(C_t) \cup \alpha_t(M_{\alpha_t}) \cup (F^t \setminus M_{\alpha_t})$) form a code assignment.
4. every node in the domain M_{α_t} of α_t is contained in $\text{dep}_t(\phi_t(C_t))$ (i.e., no unnecessary arcs are in α_t).

Note that ϕ_t and α_t can equivalently be viewed as functions and as collections of arcs of the form $(c, \phi_t(c))$ and $(u, \alpha_t(u))$, respectively. We write $\text{dep}_t(\phi_t(C_t))$ for the set $\bigcup_{c \in C_t} \text{dep}_t(\phi_t(c))$. Note that for two given independent positions $u, v \in T$ and for an independent mapping (ϕ_t, α_t) we have $\text{dep}_t(u) \cap \text{dep}_t(v) = \emptyset$, i.e. the α_t-arcs of independent positions point into disjoint subtrees. Note also that if a pair (ϕ_t, α_t) is an independent mapping for T^t, then $\text{dep}_t(\phi_t(C_t))$ is contained in opt-trees and every node in $\text{dep}_t(\phi_t(C_t))$ 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:*

$$\textit{There is an independent mapping } (\phi_t, \alpha_t) \textit{ for } T^t. \tag{6.1}$$

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. $\square$

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 cost 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 $(\phi_{t_l}, \alpha_{t_l})$, 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 $\phi_{t_l}(c)$ (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. $\square$

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 $\phi_0(c_0) = x_0$ 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 $(\phi_{t+1}, \alpha_{t+1})$ 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 $\phi_t(c_t') = g$. 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 $\phi_t(c_t) = g$. We set

$$\phi_{t+1} = \{(c, \phi_t(c)) \mid c \in C_t \setminus \{c_t\}\} \cup \{(c, \alpha_t(c)) \mid c \in \Gamma(T_g^t)\}$$

and

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

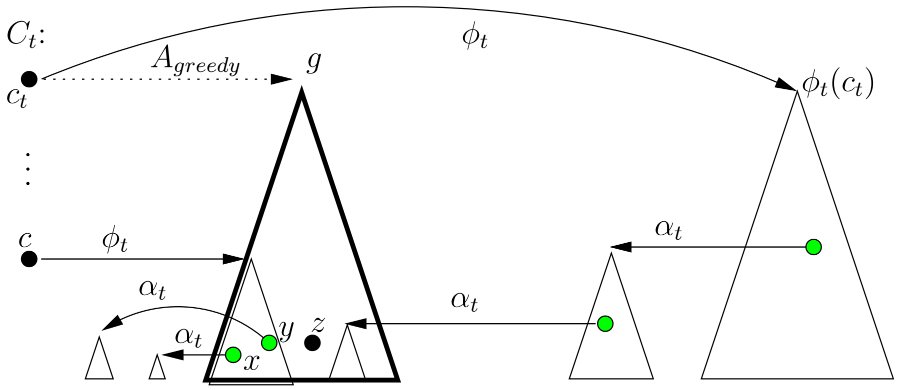


Figure 6.4: T_g^t contains the heads of α_t-arcs and ϕ_t-arcs as well as codes with and without α_t-arcs.

It is easy to see that $(\phi_{t+1}, \alpha_{t+1})$ 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 $\phi_t(c'_t) = g$. 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 $(\phi_{t+1}, \alpha_{t+1})$.

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 $\text{dep}(\phi_t(c_t))$ 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 $(\phi_{t+1}, \alpha_{t+1})$.

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 $\phi(c) = v$. For each assigned code c in T_g^t that has an α_t-arc, define $\phi(c) = \alpha_t(c)$. For all codes c in $C_t \setminus \{c_t\}$, set $\phi(c) = \phi_t(c)$. Let $\alpha = \alpha_t \setminus \{(u, v) \mid u \in T_g^t\}$. Finally, replace every α-arc or ϕ-arc (u, v) for which $v \in T_g^t$ by a loose end, i.e., an α-arc or ϕ-arc pointing from u to a dummy tree of height $l(v)$. The generalized mapping (ϕ, α) constructed in this way is indeed independent. Figure 6.5 shows the generalized mapping (ϕ, α) resulting from the situation in Figure 6.4.

Dummy trees that can be reached from $\phi_t(c_t)$ 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 on or will be discarded in the end. Similarly, we call all α-arcs that can be reached from $\phi_t(c_t)$ along tree-arcs and α-arcs *inactive*, and all

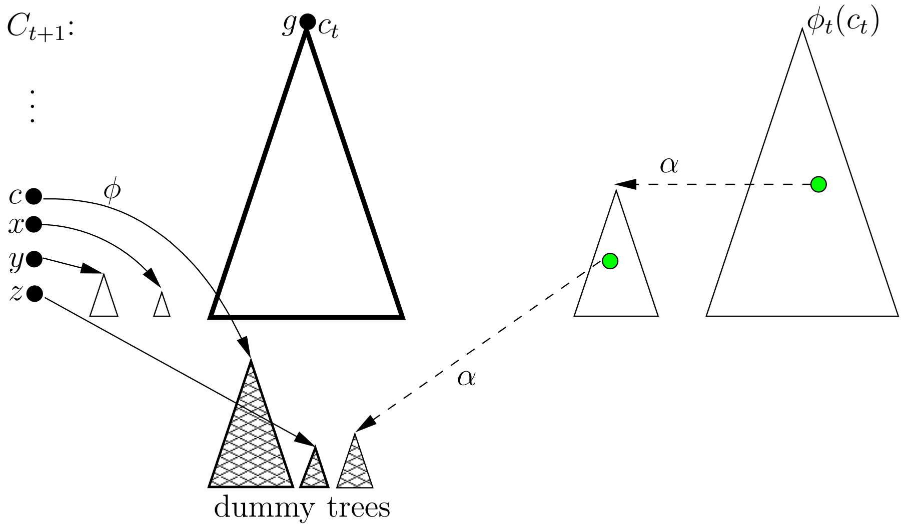


Figure 6.5: The constructed generalized independent mapping (ϕ, α). All shown α-arcs are inactive (indicated by dashed lines). The rightmost dummy tree is inactive, the other two are active.

other α-arcs *active*. Inactive α-arcs will either become active later on or will be discarded in the end as well.

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 U_i be the total capacity of inactive dummy trees. We have $U_a + U_i \leq U$.

We want to use $\mathrm{dep}(\phi_t(c_t))$ for finding new destinations for α-arcs or ϕ-arcs that point to dummy trees. We say that a path from $\phi_t(c_t)$ 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 $\mathrm{dep}(\phi_t(c_t))$ to be the number of leaves that are not in dummy trees and that can be reached from $\phi_t(c_t)$ along a strict path that does not contain the head of any ϕ-arc or active α-arc. Note that a position v in $\mathrm{dep}(\phi_t(c_t))$ 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 $\mathrm{dep}(v)$ or to a position p such that v is in $\mathrm{dep}(p)$. Otherwise, the position v is called *unavailable*.

The available capacity in $\mathrm{dep}(\phi_t(c_t))$ is $U - U_i$ initially, since only the loose ends in $\mathrm{dep}(\phi_t(c_t))$ 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 $\mathrm{dep}(\phi_t(c_t))$.

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 $\phi_t(c_t)$ 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 $\phi_t(c_t)$ 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 $\text{dep}(\phi_t(c_t))$ 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 $\text{dep}(\phi_t(c_t))$ decreases by $2^d - U'$ as well (since the part of $\text{dep}(\phi_t(c_t))$ 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 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.

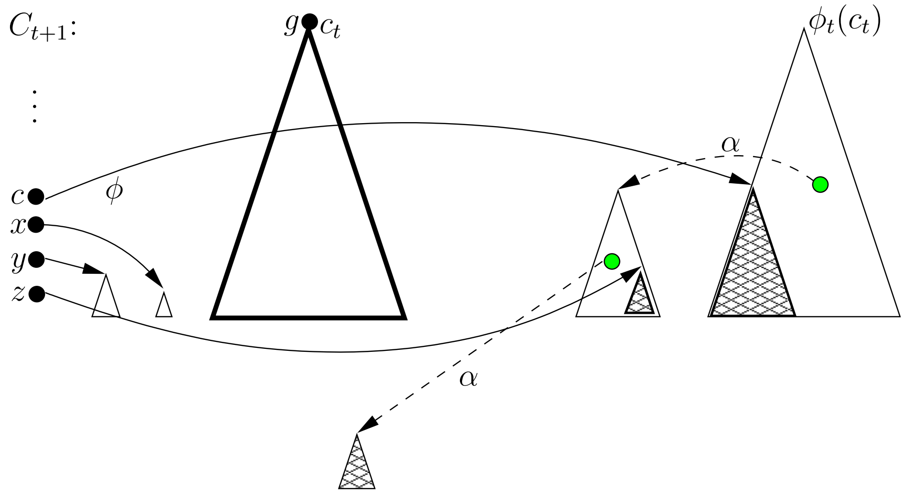


Figure 6.6: The final generalized independent mapping (ϕ, α) in which all active dummy trees have been fixed. The independent mapping $(\phi_{t+1}, \alpha_{t+1})$ is obtained by deleting the inactive α-arcs and discarding the remaining inactive dummy tree.

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

7 Online code assignment

Here we study the online CA problem. Recall that we assume that the requests never exceed the total available bandwidth. We give a lower bound on the competitive ratio,

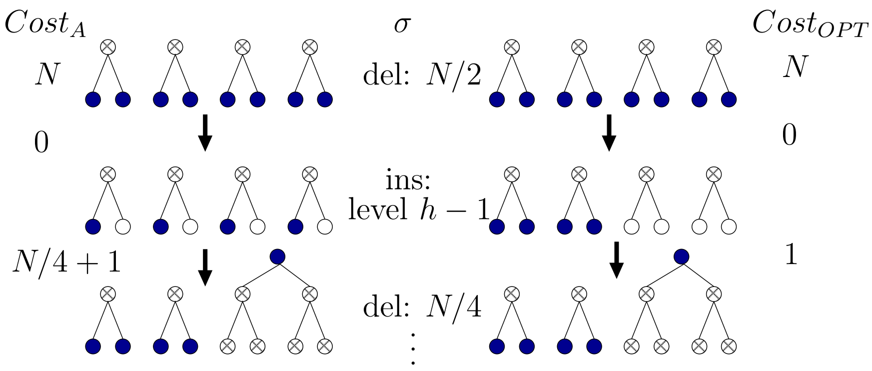


Figure 7.1: Lower bound for the online assignment problem.

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 to the situation in Figure 7.1. Then a request for a code assignment on level $h-1$ causes $N/4$ code reassignments. We can proceed with the left subtree of full leaf codes recursively. We can repeat this process $(\log_2 N - 1)$ times. The optimal algorithm A_{opt} assigns the leaves 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, $T(N) = \log_2 N - 1 + \frac{N}{2}(1 - 2/N)$. If $C_A \leq c \cdot C_{OPT}$ then $c \geq \frac{3N/2 + \log_2 N - 2}{N + \log_2 N - 1} \longrightarrow_{N\to\infty} 3/2$. □

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 $w(v)$ its string representation, i.e. the description of the path from the root to the node/code, where 0 means left child and 1 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:

$$\forall \text{ codes } u, v \in F: \; l(u) < l(v) \Rightarrow w(u) < w(v), \tag{7.1}$$

$$\forall \text{ nodes } u, v \in T: \; l(u) \leq l(v) \;\wedge\; u \in F \;\wedge\; v \in U \;\Rightarrow\; w(u) < w(v). \tag{7.2}$$

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).

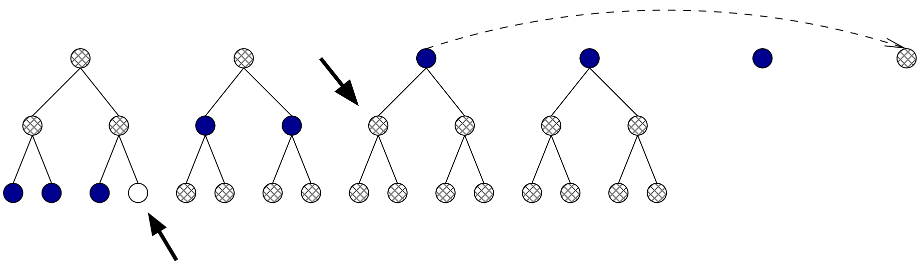


Figure 7.2: Algorithm $A_{compact}$ finds the rightmost position (blocked or unblocked) for a code insertion and reassigns at most one code at every level.

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 $\mathcal{O}(h)$ code reassignments per request.*

Proof. We show that for both insertion and deletion requests 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 assignment request for 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 request.

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. *The 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. □

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

Proof. Consider the sequence of requests for code assignments on levels $0, 0, 1, 2, 3, \ldots, h-1$. 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 zero and maintain the invariant (7.1). This takes $h-1$ code (re-)assignments. Consider as the next requests the deletion of the third code on level zero and an insertion

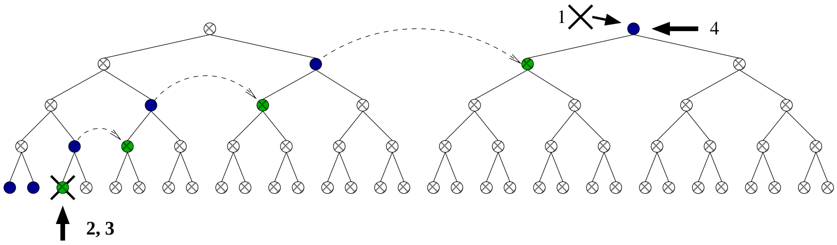


Figure 7.3: Code assignments for levels $0, 0, 1, 2, 3, 4, \ldots, h-1$ and four consecutive operations: 1. DELETE(h-1), 2. INSERT(0), 3. DELETE(0), 4. INSERT(h-1).

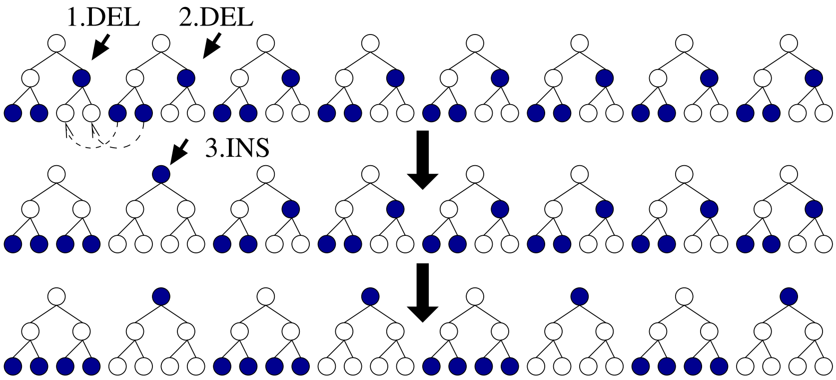


Figure 7.4: Requests that a greedy strategy cannot handle efficiently.

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 $h-1$ code (re-)assignments. 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 the algorithm A_I is then $C_A = h + k \cdot (2h-2)$, whereas OPT has $C_{OPT} = h + k \cdot 2$. As k goes to infinity, the ratio C_A/C_{OPT} is $\Omega(h)$. □

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 request, 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

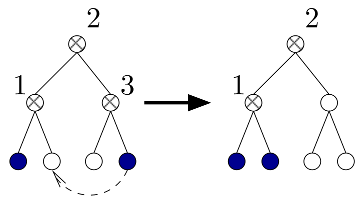


Figure 7.5: Reassignment of one code reduces the number of blocked codes from 3 to 2.

assign a code at level $l+1$. As it is optimal (and up to symmetry unique) the 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 $N/2^{i+2}$ new codes on level $i+1$. Altogether the greedy strategy needs $\mathcal{O}(N) + (N/4)\Omega(\log N) = \Omega(N \log N)$ (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 [18] but not analyzed at all. In every step the algorithm tries to satisfy the invariant:

$$\#\text{ blocked codes in } T \text{ is minimum.} \tag{7.3}$$

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.

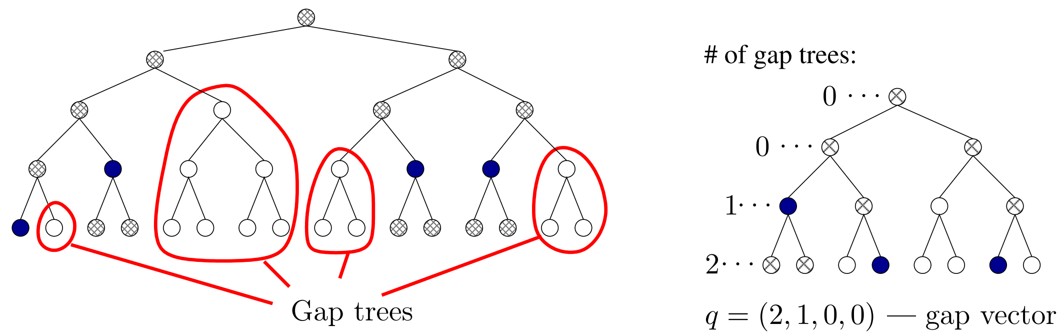
We can prove that this approach is equivalent to minimizing the number of *gap trees* on every level (Theorem 7.7).

Definition 7.6. A maximal subtree of unblocked codes is called a *gap tree*. The level of its root is called the *level of the gap tree*. The vector $q = (q_0, \ldots, q_h)$, $q_i =\#$ gap trees on level i, is called the *gap vector* of the tree T.

We can see that the invariant (7.3) implies at most one gap tree at each level. For example in Figure 7.6(a) we have two gap trees on level one. Using the concept from Figure 7.5, i.e. moving a sibling tree of the gap tree to fill the second gap tree, we reduce the number of blocked codes by at least one.

Theorem 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.*

Proof. Suppose T has minimum number of blocked codes. If T had two gap trees T_u, T_v on level l, then we could move the codes in sibling tree $T_{u'}$ (there are some) of u into T_v,



(a) Gap trees—maximal subtrees of unblocked codes

(b) Gap vector

Figure 7.6: Definition of gap trees and gap vector

which would save at least one blocked code (the parent of u would become unblocked), a contradiction.

Now 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 thus 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. $\square$

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

Algorithm 7.8. *A_{gap}:*

1. *Insert:* • *Assign the new code into the smallest gap where it fits.*
2. *Delete:* • *If after the deletion a second gap tree appears on some level, move one of their sibling subtrees to "fill" the gap tree*
   • *Look for a second gap tree on a higher level and treat it recursively.*

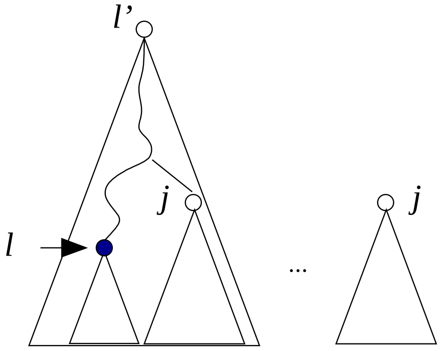


Figure 7.7: Two gap trees on a lower level than l' violate the minimum chosen height of the gap tree.

Lemma 7.9. *The algorithm A_{gap} has always 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.*

Proof. We know that there is sufficient capacity to serve the request, i.e. $cap \geq 2^l$. We also know that $cap = \sum_i q_i 2^i$. Summing only over gap trees on level $i < l$ we get a capacity $cap' = \sum_{i=0}^{l-1} q_i 2^i \leq 2^0 + 2^1 + \ldots 2^{l-1} = 2^l - 1$. Therefore, there exists a gap tree on level $j \geq l$.

Next, consider an insertion operation into the smallest gap tree on 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, a contradiction (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[7]. Thus, when the newly created gap tree is the second one at the level, we fill the gap trees and then we recursively handle the newly created gap tree at 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 at 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 $\Omega(\log N)$-competitive.*

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

[7] and some gap trees may disappear

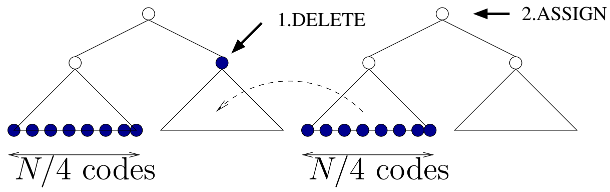


Figure 7.8: Worst case number of movements for algorithm A_{gap}.

The 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 *3-gap* and study it by a resource-augmented competitive analysis. The strategy *3-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. *3-gap* uses only an amortized constant number of reassignments per insertion or deletion.

3-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 3 gaps (unblocked codes) 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 2 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 reassign the rightmost code on level ℓ to c, keeping all codes at level ℓ left of the gaps of ℓ. If this results in 4 consecutive gaps, we reassign the rightmost code of $\ell+1$, 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 there are at most 3 gaps allowed. If we run out of space or if there are too many gaps, we move the boundary between two consecutive levels, affecting two places on the lower level and one on the upper level. This notion of a range is in particular important for levels without codes. The levels close to the root are handled differently, to avoid an excessive space usage. The root code of T' has bandwidth $2b$, it is never used. The bandwidth b code can only be used if no other code is used, there is no interaction with other codes. The $b/2$ codes 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. For all other levels ($\leq b/8$ codes) we define a potential-function by counting the number of levels without gaps, and the number of levels having 3 gaps, and adding these numbers. 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).

It remains to show that our algorithm manages to host codes as long as the total bandwidth used does not exceed b. To do this, we calculate the bandwidth wasted by gaps, which is at most $3(\frac{b}{8} + \frac{b}{16} + \cdots) \leq 3b/4$. Hence the total bandwidth used in T' is $7b/4 < 2b$.

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 the bandwidth is exceeded. Then the above online strategy uses a code tree of height $h+1$ and performs at most $2m$ code assignments and reassignments.*

Corollary 7.12. *The above strategy is 4-competitive for resource augmentation by a factor of 2.*

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 $2m$ (re-)assignments, leading to a competitive factor of 4. □

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

8 Enforcing arbitrary configurations

We have discussed already the hardness of the one-step offline problem. We constructed a special configuration of the code tree in order to encode an instance of another hard problem, $3D$-matching. One can wonder whether this configuration would ever be attained when an optimal one-step offline algorithm is applied to a sequence of requests. 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. The result even holds for any algorithm A that only reassigns codes if it has to, i.e. it places a code without any additional reassignments if this is possible and does not reassign after a deletion.

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 arbitrary configuration C_{full} that uses the full tree capacity.

Theorem 8.1. *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. We proceed top-down: On every level l' with codes in C_{full} we first fill all its unblocked positions using at most $2^{h-l'}$ requests of level l'. A just fills l' with codes. Then we delete all codes on l' that are not in C_{full} and proceed recursively on 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 operation 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. □

Now we come back to arbitrary configurations.

Corollary 8.2. *Given a configuration tree C' of height h with n assigned codes, there exists a sequence $\sigma_1, \ldots, \sigma_m$ of code requests of length $m < 4nh$ that forces A into C'.*

Proof. To go from C' to C_{full} we fill 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 8.1, 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. □

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. To our knowledge, we are the first to analyze the computational complexity of the OVSF code assignment problem. We point out that the algorithm in [17], 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. The merits of this paper, besides the rigorous analyses and results presented, lie in the challenge addressed to theoreticians to solve the open problems raised by our work:

- 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?
- What is the complexity of the general offline CA problem?
- Is there a fixed-parameter tractable [8] algorithm for the one-step offline CA problem with at most k reassignments?

References

[1] J. Adamek. *Foundation of Coding.* John 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 *Proceedings of the IEEE International Conference on Communications (ICC)*, 2002.

[5] H. Çam. Nonblocking OVSF codes and enhancing network capacity for 3G wireless and beyond systems. *Computer Communications*, 2003. To appear.

[6] J.-C. Chen and W.-S. E. Chen. Implementation of an efficient channelization code assignment algorithm in 3G WCDMA. National Chung Hsing University, manuscript, 2001.

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

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

[9] 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.

[10] Amos Fiat and Gerhard J. Woeginger, editors. *Online Algorithms: The State of the Art.* LNCS 1442. Springer-Verlag, Berlin, 1998.

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

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

[13] M. R. Garey and D. S. Johnson. *Computers and Intractability*. Freeman, 1979.

[14] H. Holma and A. Toskala. *WCDMA for UMTS*. Wiley, 2001.

[15] 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.

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

[17] 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.

[18] 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 13th IEEE International Symposium on Personal, Indoor and Mobile Radio Communications*, 2002.

A Exact algorithm with example

Algorithm A.1 shows the pseudo-code for the dynamic programming approach presented in Section 5. For a better understanding of the algorithm we give an example for the original tree in Figure A.1 and a new code request at level two.

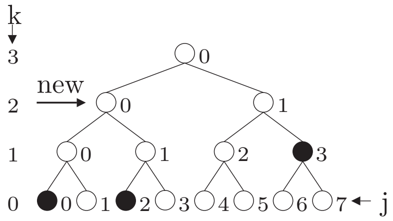


Figure A.1: Original tree and the new code request

The tables that the algorithm constructs are shown in Table 1. For the root level M_3 it is enough to consider only the final tree signature. The reconstruction of the final tree (which is performed by the function call to *ReconstructTree* in Algorithm A.1) starts with the table M_3, takes the final signature for the root node, and goes recursively following the left and right pointers into the table M_2. The final tree configuration of the example can be seen in Figure A.2.

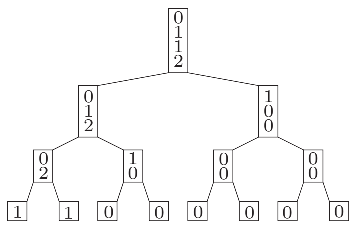


Figure A.2: Final tree configuration for the example in Figure A.1

Algorithm A.1. *Optimal Algorithm A_{OPT}*
in: *T_h original tree configuration; l level of the new code*
out: *T'_h optimal final tree configuration*
params: *h height of the tree; M_k table for level k; S_k temporary signature for level k; $M_k[S_k, j]$ entry in table M_k with the following fields: maximum profit P, left tree signature L, right tree signature R; lj and rj are the column indices for the left and right subtrees of node j*

```
begin
  for k ← 0 to h do
    Allocate(M_k);InitProfit(M_k,0)
  end for
  Initialize(M_0)
  for k ← 1 to h − 1 do
    {iterate over all nodes of level k}
    for j ← 0 to 2^(h−k) − 1 do
      {fill in column j}
      for all signatures S_left in M_(k−1) do
        for all signatures S_right in M_(k−1) do
          S_k ← S_left + S_right augmented with a 0 at position k
          P ← M_(k−1)[S_left, lj].P + M_(k−1)[S_right, rj].P
          if M_k[S_k, j].P < P then
            M_k[S_k, j].P ← P
            M_k[S_k, j].L ← S_left
            M_k[S_k, j].R ← S_right
          end if
        end for
      end for
      if node j originally had a code assigned to it then
        S_k ← [1, 0, . . . , 0, 0]^T
        M_k[S_k, j].P ← 1
        M_k[S_k, j].L ← the all zero signature from col lj of M_(k−1)
        M_k[S_k, j].R ← the all zero signature from col rj of M_(k−1)
      end if
    end for
  end for
  FillIn(M_h[V'_h, 0])
  T'_h ← ReconstructTree(M_h[V'_h, 0])
  return T'_h
end A_OPT
```

Table 1: Tables constructed by Algorithm A.1 for the example in Figure A.1

M_0	0	1	2	3	4	5	6	7
$\begin{bmatrix} 0 \end{bmatrix}$	0	0	0	0	0	0	0	0
$\begin{bmatrix} 1 \end{bmatrix}$	1	0	1	0	0	0	0	0

M_1	0	1	2	3
$\begin{bmatrix} 0 \\ 0 \end{bmatrix}$	$0, \begin{bmatrix} 0 \end{bmatrix}_l, \begin{bmatrix} 0 \end{bmatrix}_r$	$0, \begin{bmatrix} 0 \end{bmatrix}_l, \begin{bmatrix} 0 \end{bmatrix}_r$	$0, \begin{bmatrix} 0 \end{bmatrix}_l, \begin{bmatrix} 0 \end{bmatrix}_r$	$0, \begin{bmatrix} 0 \end{bmatrix}_l, \begin{bmatrix} 0 \end{bmatrix}_r$
$\begin{bmatrix} 0 \\ 1 \end{bmatrix}$	$1, \begin{bmatrix} 1 \end{bmatrix}_l, \begin{bmatrix} 0 \end{bmatrix}_r$	$1, \begin{bmatrix} 1 \end{bmatrix}_l, \begin{bmatrix} 0 \end{bmatrix}_r$	$0, \begin{bmatrix} 0 \end{bmatrix}_l, \begin{bmatrix} 1 \end{bmatrix}_r$	$0, \begin{bmatrix} 1 \end{bmatrix}_l, \begin{bmatrix} 0 \end{bmatrix}_r$
$\begin{bmatrix} 0 \\ 2 \end{bmatrix}$	$1, \begin{bmatrix} 1 \end{bmatrix}_l, \begin{bmatrix} 1 \end{bmatrix}_r$	$1, \begin{bmatrix} 1 \end{bmatrix}_l, \begin{bmatrix} 1 \end{bmatrix}_r$	$0, \begin{bmatrix} 1 \end{bmatrix}_l, \begin{bmatrix} 1 \end{bmatrix}_r$	$0, \begin{bmatrix} 1 \end{bmatrix}_l, \begin{bmatrix} 1 \end{bmatrix}_r$
$\begin{bmatrix} 1 \\ 0 \end{bmatrix}$	$0, \begin{bmatrix} 0 \end{bmatrix}_l, \begin{bmatrix} 0 \end{bmatrix}_r$	$0, \begin{bmatrix} 0 \end{bmatrix}_l, \begin{bmatrix} 0 \end{bmatrix}_r$	$0, \begin{bmatrix} 0 \end{bmatrix}_l, \begin{bmatrix} 0 \end{bmatrix}_r$	$1, \begin{bmatrix} 0 \end{bmatrix}_l, \begin{bmatrix} 0 \end{bmatrix}_r$

M_2	0	1
$\begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$	$0, \begin{bmatrix} 0 \\ 0 \end{bmatrix}_l, \begin{bmatrix} 0 \\ 0 \end{bmatrix}_r$	$0, \begin{bmatrix} 0 \\ 0 \end{bmatrix}_l, \begin{bmatrix} 0 \\ 0 \end{bmatrix}_r$
$\begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$	$1, \begin{bmatrix} 0 \\ 1 \end{bmatrix}_l, \begin{bmatrix} 0 \\ 0 \end{bmatrix}_r$	$0, \begin{bmatrix} 0 \\ 1 \end{bmatrix}_l, \begin{bmatrix} 0 \\ 0 \end{bmatrix}_r$
$\begin{bmatrix} 0 \\ 0 \\ 2 \end{bmatrix}$	$2, \begin{bmatrix} 0 \\ 1 \end{bmatrix}_l, \begin{bmatrix} 0 \\ 1 \end{bmatrix}_r$	$0, \begin{bmatrix} 0 \\ 2 \end{bmatrix}_l, \begin{bmatrix} 0 \\ 0 \end{bmatrix}_r$
$\begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}$	$0, \begin{bmatrix} 1 \\ 0 \end{bmatrix}_l, \begin{bmatrix} 0 \\ 0 \end{bmatrix}_r$	$1, \begin{bmatrix} 0 \\ 0 \end{bmatrix}_l, \begin{bmatrix} 1 \\ 0 \end{bmatrix}_r$
$\begin{bmatrix} 0 \\ 1 \\ 1 \end{bmatrix}$	$1, \begin{bmatrix} 0 \\ 1 \end{bmatrix}_l, \begin{bmatrix} 1 \\ 0 \end{bmatrix}_r$	$1, \begin{bmatrix} 0 \\ 1 \end{bmatrix}_l, \begin{bmatrix} 1 \\ 0 \end{bmatrix}_r$
$\begin{bmatrix} 0 \\ 1 \\ 2 \end{bmatrix}$	$1, \begin{bmatrix} 0 \\ 2 \end{bmatrix}_l, \begin{bmatrix} 1 \\ 0 \end{bmatrix}_r$	$1, \begin{bmatrix} 0 \\ 2 \end{bmatrix}_l, \begin{bmatrix} 1 \\ 0 \end{bmatrix}_r$
$\begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$	$0, \begin{bmatrix} 0 \\ 0 \end{bmatrix}_l, \begin{bmatrix} 0 \\ 0 \end{bmatrix}_r$	$0, \begin{bmatrix} 0 \\ 0 \end{bmatrix}_l, \begin{bmatrix} 0 \\ 0 \end{bmatrix}_r$

M_3	0
$\begin{bmatrix} 0 \\ 1 \\ 1 \\ 2 \end{bmatrix}$	$1, \begin{bmatrix} 0 \\ 1 \\ 2 \end{bmatrix}_l, \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}_r$