A Quasi Optimal Bit-vector Encoding of Tree Hierarchies.
Application to Efficient Type Inclusion Tests.

Olivier Raynaud and Eric Thierry

LIRMM, 161 rue Ada, 34392 Montpellier Cedex 5, France
raynaud@lirmm.fr,thierry@lirmm.fr

Abstract. Type inclusion tests consist in determining whether a type is a subtype of another. An efficient implementation of type inclusion is an important feature of object oriented programming languages.
A well-known method to achieve these tests is to associate to each type a subset of a set $S = \{1, \ldots, k\}$ such that type inclusion coincides with subset inclusion. Such an embedding of types into 2^S (the lattice of all subsets of S) is called a bit-vector encoding of the type hierarchy. These encodings are known for several interesting features. First bit-vector encodings are perfectly appropriate for hierarchies with single subtyping as well as hierarchies with multiple subtyping. Secondly subset inclusion tests can be performed very efficiently. Thirdly they admit incremental algorithms which can modify the encoding at low costs when the hierarchy evolves. Finally some of them enable to compute efficiently greatest lower bounds or least upper bounds when they exist ([1]).
Several works have studied bit-vector encodings from a theoretical point of view ([6, 7, 10, 13, 15, 17, 18]) and from a practical point of view ([3, 8, 9, 16]), in particular in order to minimize the size of the encoding, i.e. the size of S.
In this article, we present a new algorithm which computes bit-vector encodings for single subtyping hierarchies, also called tree hierarchies. This particular case of the tree hierarchies has already been studied in the works of Caseau ([3, 18]) and Krall, Vitek and Horspool ([9, 16]).
Our algorithm is simple, it computes the bit-vector encoding very quickly and gives good results for the size of the encoding. In particular, we have significantly improved the best bounds known for the encoding sizes of some benchmarks presented in [9]. We will see that thanks to its efficiency, this encoding can compete with previous kinds of encodings which were specifically designed for tree hierarchies. Moreover it can take advantage of the incremental aspects of bit-vectors encodings. As an example, we propose a new strategy to encode multiple subtyping hierarchies that are almost tree hierarchies (with little use of multiple subtyping).

Keywords: Type inclusion, subsumption, partial order, bit-vector encoding, compression of hierarchy,tree hierarchy.

1 Introduction

Type inclusion testing has an important role in modern programming languages, in particular for object oriented languages where some programs frequently ask whether a type is a subtype of another.

The type inclusion relation (or subtype relation) is a transitive, reflexive anti-symmetric relation, thus it is a partial order on the types of the program, also called the hierarchy of types. In order to check the inclusion of types, many techniques called ***hierarchical encodings*** have been developed [1, 3–6, 9]. Figure 1 represents such an encoding for the Petra hierarchy. Petra is a generic application for the modelisation of telephone networks. The usual requirements for these encodings are the runtime efficiency, so that type tests should be fast, and the minimization of the space used to store the encoding.

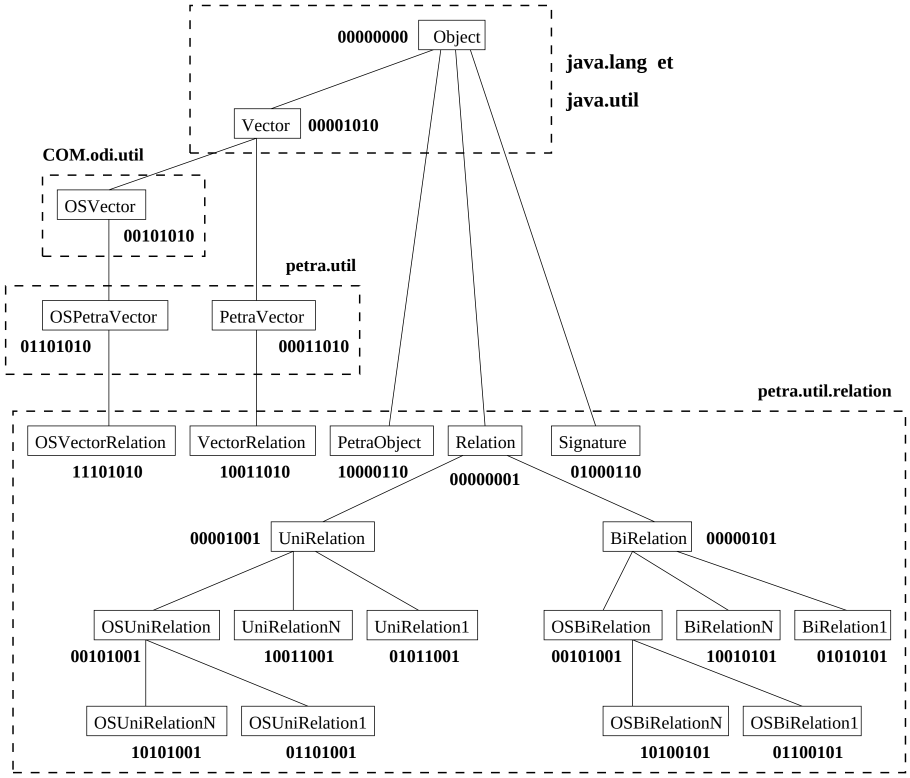


Fig. 1. Class hierarchy of Petra relations

A classical way to encode a hierarchy of N types is to index these types with integers from 1 to N and store the subtype relation in a binary matrix M of size $N \times N$ where $M[x, y] = 1$ if the type indexed by y is a subtype on the one indexed by x, and $M[x, y] = 0$ otherwise. With this method, type inclusion tests can be performed in constant time. But the main drawback is the quadratic space needed to store the matrix.

Another method consists in associating with each type the list of its direct subtypes (a direct subtype of A is a subtype B of A such that there is no other type between A and B for the subtype relation). This method which saves space is however slow for checking whether a type is a subtype of another.

More generally it is difficult to optimize both speed and space, and it is necessary to choose an encoding with a good compromise, depending of the needs (see for instance [16] for an evaluation of several kinds of encodings).

Another important point is the ability to recompute an encoding of the hierarchy when this one evolves, for instance when a new type is added. When the encoding algorithm of the whole hierarchy is computationally intensive or when the hierarchy is frequently modified, it is useful to have a incremental algorithm which computes a new encoding directly from the previous encoding after each modification (and do not recompute it from the beginning).

In this article, we will study a well-known kind of hierarchical encoding, called ***bit-vector encoding***. The idea is to associate with each type a subset of a fixed set $S = \{1, \ldots, k\}$ such that type inclusion coincides with subset inclusion. More formally, we denote by $P = (X, \leq)$ the type hierarchy where X is the set of the types and $x \leq y$ if and only if y is a subtype of x. A ***bit-vector encoding*** of P is an application ϕ from X into 2^S (the lattice of all the subsets of S) such that: $x \leq y$ if and only if $\phi(x) \subseteq \phi(y)$. The ***size*** of the encoding ϕ is the cardinal of S. The elements of S are sometimes called ***colors*** or ***genes*** ([3]).

In order to describe the hierarchies, we also introduce the next definitions. Let x be a type in the hierarchy $P = (X, \leq)$, $Anc(x) = \{y \in X \mid y \neq x,\ y \leq x\}$ is the set of the ***ancestors*** of x and $Desc(x) = \{y \in X \mid y \neq x,\ x \leq y\}$ is the set of the ***descendants*** of x. A direct subtype y of the type x is called a ***child*** of x and x is a ***parent*** of y. A ***chain*** is a sequence $x_0, x_1, \ldots, x_h$ of types such that $x_0 \leq x_1 \leq \ldots \leq x_h$, the ***height*** of this chain is h. The ***height*** of the hierarchy P is the maximal height of all the chains of P, it is denoted by $height(P)$. The ***degree*** of a type x is the number of direct subtypes of x, it is denoted by $deg(x)$.

Conventionally the hierarchies will be drawn in a top-down fashion. The edges represent the direct subtype relation with the subtypes below their parent types. Figure 2 shows three different representations of a unique bit-vector encoding ϕ of size $|S| = 6$ for a given hierarchy P. The first one associates to each type x its code $\phi(x)$ which is a subset of $S = \{1, 2, 3, 4, 5, 6\}$. The second one associates with each type x its ***reduced code*** $\phi_r(x)$ defined by $\phi_r(x) = \phi(x) \setminus \bigcup_{y \in Anc(x)} \phi(y)$. The last one associates to each type x an vector of $|S|$ bits where bit i is equal to 1 if $i \in \phi(x)$ and equal to 0 otherwise. This is a way to implement bit-vector encodings and it uses $|S|$ bits per type. These three representations are perfectly equivalent. The complete code is obtained from the reduced code by propagating the colors of the types to their descendants. The encoding of Figure 1 represents a bit-vector encoding of the Petra hierarchy.

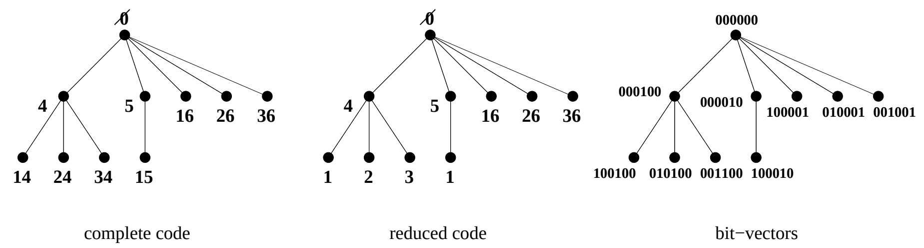


Fig. 2. The three different representations of a bit-vector encoding

Several known encoding methods compute bit-vector encodings [3, 9, 12, 18]. We enumerate below the strong points of bit-vector encodings:

1. Appropriate for single subtype hierarchies and multiple subtype hierarchies.
2. Economical with space (good compression compared to the binary matrix encoding).
3. Fast type inclusion tests (constant time for bit-vectors of bounded size)
4. Lattice operations (lowest upper bound or greatest lower bound when they exist) in constant time with special bit-vector encodings.
5. Existence of incremental algorithms.

A ***single subtyping hierarchy*** is a hierarchy where each type has at most one parent. The other hierarchies are called ***multiple subtyping hierarchies***. Bit-vector encodings are appropriate for single subtyping hierarchies as well as multiple subtyping hierarchies. As it will be seen in Section 2, it is not always the case for hierarchical encodings. Some encodings especially designed for single subtyping hierarchies do not extend to multiple subtyping hierarchies.

For a given hierarchy, the smallest bit-vector encodings are very compact encodings with sizes usually closer to $\log_2(N)$ bits rather than N bits (where N is the number of types in the hierarchy). It represents a very important saving of space in comparison with the binary matrix storage which requires N bits per type. Unfortunately finding the minimum size of a bit-vector encoding for a given hierarchy (a parameter which is also called the ***2-dimension*** of the hierarchy [2, 15]) is known to be a NP-hard problem [14] and recently it has been shown that approximating the 2-dimension is equivalent to approximating the optimal coloration of a graph [12]. A consequence of this result is that there does not exist an algorithm which approximates the 2-dimension up to a constant factor unless P=NP. Fortunately, it is possible to propose heuristics that produce very compact bit-vector encodings for practical cases. It is actually the aim of this article.

Concerning the speed of the type inclusion tests, checking whether $x \leq y$ is equivalent to check whether $\phi(x)$ OR $\phi(y) = \phi(y)$. These are elementary boolean operations and if we consider that the size of the bit-vectors is bounded (for instance if we known that it can be stored on two machine words), it is a constant time test (for a discussion about "constant time" tests, see [16] where several techniques of tests are presented with their implementation using RISC instructions).

These bit-vector encodings can also have interesting lattice properties. Some of them enable to compute very efficiently (in "constant time") lowest upper bounds (resp. greastest lower bounds) when they exist. For a study of this more specific problem, see [1] and [10].

If we add a new type x in an encoded hierarchy, it is possible to build a new encoding immediately: we add a new color to S and we add this color to the code of x and all its descendants. This is a fast modification but costly for space after several additions. Fortunately in order to keep minimizing the storage space, there exists an incremental algorithm [13, 18]. It is based on the coloring of a conflict graph where most of the vertices are already efficiently colored. These incremental aspects will be reexplained in Section 5.

In this article, we present a new algorithm to compute a near optimal bit-vector encoding for single subtyping hierarchies. Without loss of generality, we only consider ***tree hierarchies*** which are single subtyping hierarchies with a root type (a top element). The types with no subtype are called leaves. Such hierarchies of types often occur in object oriented languages, even in the ones that allow multiple subtyping. The motivation of this study is to improve the size bounds for people using the framework of bit-vector encodings. We show how to take into account the tree structure in order to minimize the encoding size. Moreover, hierarchies are frequently almost tree hierarchies. In this case, it is possible to construct an encoding based on the encoding of a large tree subhierarchy, modified to encode the whole hierarchy (for instance with an incremental algorithm).

In Section 2, we give a short overview of the previous works dealing with tree hierarchies. We present bit-vector encoding algorithms as well as other encoding algorithms for comparison. Our algorithm is described in Section 3. This is a simple greedy algorithm with no preprocessing step and no setting of parameters for optimization. The Section 4 evaluates the performances of this algorithm. We show that we have obtained the optimal balancing strategy for trees. Finding this optimal balancing was conjectured to be feasible for trees in [9]. As a result, we have significantly improved the encoding sizes of some benchmarks coming from [9]. This section contains a comparison of the encoding sizes of known algorithms on these benchmarks and on generic tree hierarchies (complete trees and comb trees). Incremental aspects are tackled in Section 5 when we suggest a strategy to encode hierarchies that are almost tree hierarchies.

2 Previous works

Many techniques are known to achieve type inclusion tests for tree hierarchies. First we must present a well-known encoding which is optimal in time and space. It is not a bit-vector encoding, this algorithm finds a good numbering of the types. For each type x of the hierarchy, a couple of integers $(l(x), r(x))$ are computed as follows: traverse the tree with a Depth First Search and for each new type increment a counter c. When a type x is first encountered, store c in $l(x)$. When the traversal leaves the type store the current value of c in $r(x)$. Then a type x is a subtype of a type y if and only if $l(y) \leq l(x)$ and $r(x) \leq r(y)$. In order to store an encoding of N types, we can say that this encoding uses $2\lceil \log_2(N) \rceil$ bits per type and a type inclusion test just requires two integer comparisons (constant time comparisons). Figure 3 shows a example of such an encoding. Unfortunately this method can not be extended to multiple subtyping hierarchies and if the tree is modified, it is not easy to recompute a new encoding apart from recomputing the whole encoding from the beginning.

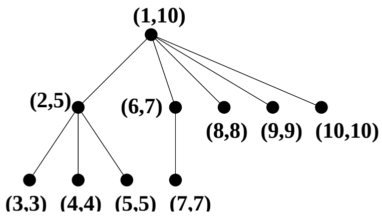


Fig. 3. hierarchical encoding thanks to a good numbering of types

Among the studies about tree hierarchies, we can mention the Cohen's algorithm [11] which proposes constant type tests and uses a matrix $N \times H$ of integers where H is the height of the hierarchy. It represents $H * \log_2(N)$ bits per type, it is usually better than the binary matrix method but it still may be costly in space, especially compared to bit-vector encodings.

One naive method to compute a bit-vector encoding associates with each type a different index and consider this single color as the reduced bit-vector encoding. The corresponding encoding after propagation of the colors has size N where N is the number of types. It needs the same space as the binary matrix method. The space cost is extremely expensive compared to the near optimal bit-vector encodings as we will see.

Caseau is the first to have studied bit-vector encodings in the special case of tree hierarchies [3]. He focussed on a particular class of bit-vector encodings: those whose reduced encoding have at most one color per type. His method computes a conflict graph from the hierarchy then colors the graph and finally computes a bit-vector encoding with the colors used for the graph. His encoding makes a interesting reuse of colors and he was able to precise the size of its encoding for tree hierarchies. Let C be a chain $x_0, x_1, \ldots, x_p$ of a tree hierarchy T, he defines the *weight* of this chain by $weight(C) = \sum_{0 \leq i \leq p} deg(x_i)$. The size of Caseau's encoding of T is the maximum weight for all the chains of T. Figure 5 shows an example of the reduced encoding produced by this method (the chain of maximum weight is in bold).

In their article [9], Vitek, Krall and Horspool have revisited this approach. They also construct a conflict graph and compute a bit-vector encoding from a coloration of this graph. The reduced encoding also has at most one color for each type, but they start by modifying the initial hierarchy in order to get smaller bit-vector encodings. They have better results thanks to this preprocessing, but their algorithms remains tributed (tributaires) to the performance of coloring algorithms which determines the size of the encoding. They included the idea of "balancing" the hierarchy during preprocessing, we will come back to this point in Section 4.

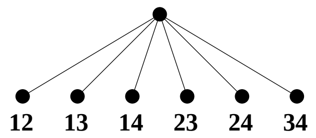


Fig. 4. The optimal bit-vector encoding for a type and its 6 children

In [18], Caseau, Habib, Nourine and Raynaud went back to the conflict graph method in order to encode any hierarchy but they proposed to use a special algorithm if the input appends to be a tree hierarchy. They present an algorithm especially designed for tree hierarchy. In order to give the size of their encoding, we have to define a fonction $cmin(n)$ which is the minimum number of colors needed to encode a type with n direct subtypes. It is known that $cmin(n) = \min\{k \mid \mathrm{C}_k^{k/2} \geq n\}$ and it is approximately $\lceil log_2(n) \rceil$ (for all $n \leq 40000$, $\lceil log_2(n) \rceil \leq cmin(n) \leq \lceil log_2(n) \rceil + 2$) . For instance $cmin(6) = 4$ as it is shown on Figure 4. We redefine a weight function for a chain C by $weight(C) = \sum_{0 \leq i \leq p} cmin(deg(x_i))$. The size of their encoding of T is the maximum of the weight function taken over all the chains of T. Figure 5 shows such an encoding. This algorithm is clearly a amelioration of Caseau's algorithm in the case of tree hierarchies.

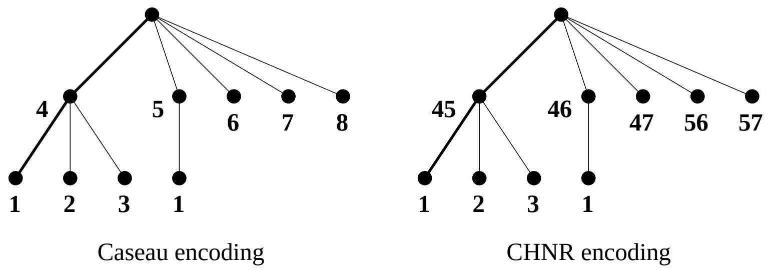


Fig. 5. Encodings generated by Caseau's method and CHNR's method

The next section presents our algorithm, it is based on two ideas that were already raised in [3] and [9]: splitting and balancing.

3 The encoding algorithm

We present here a new algorithm for computing a very compact bit-vector encoding of tree hierarchies. First in order to formalise the "splitting and balancing" strategies, we introduce a particular class of bit-vector encodings: the ***dichotomic encodings***. Then we describe our algorithm which compute a compact dichotomic encoding. As we will see in Section 4, it produces the smallest dichotomic encoding for the input tree hierarchy.

3.1 Dichotomic encodings

The idea consists in splitting in two parts the initial tree and giving a different color to the codes of each subtree. By repeating this operation on each part recursively until each subtree is empty, we are able to produce what we call a dichotomic encoding.

Let T be a hierarchy with a bit-vector encoding, let x be a color of this encoding, we will call T_x the set of the elements of T which contain x in their code. We give a recursive definition of a *dichotomic encoding* .

Definition 1. *Let T be a hierarchy, we will say that an encoding of T is dichotomic if there exist two elements x and y such that:*

- *Every element of T contains either x or y in its code.*
- *The codes of T_x minus x form a dichotomic encoding of T_x.*
- *The codes of T_y minus y form a dichotomic encoding of T_y.*
- *If T is composed by an unique vertex, the empty code is considered as a dichotomic encoding.*

Let us first remark that a dichotomic encoding is equivalent to an embedding of the hierarchy into a binary tree. We call this technique *dichotomic* because we split the tree in two differents parts which can be encoded independently. Secondly note that their exist many dichotomic encodings for a unique tree. We will show that our algorithm produces the most compact one.

3.2 The algorithm

This is a bottom-up greedy algorithm which basically constructs a binary tree from the input hierarchy T by introducing intermediate nodes (symbolising "splitting" operations) and it carries weights on the nodes in order to "balance" this binary tree. The weight of a node s corresponds to the number of colors necessary to encode the subtree composed of s and its descendants and denoted by $T(s)$. To compute this information for a node s (stored in $weight(s)$) we need to dispose of this information for every child of s.

At the beginning, only the leaves carry a weight equal to zero. Then suppose that all the children of a node s are carrying a weight, three cases can occur.

1. If s has only one child v, the subtree $T(s)$ needs $weight(v) + 1$ colors to be encoded. One color to encode the vertex v and $weight(v)$ to encode the subtree $T(v)$.
2. If s has exactly two children u and v such that $weight(u) \leq weight(v)$, we encode $T(u)$ with colors dedicated to $T(v)$ and we encode vertices u and v with two news colors. This encoding of $T(s)$ will use $weight(v) + 2$ colors.
3. If s has more than two children, to assure that our methode produce the most compact dichotomic encoding, we will marry the two subtrees $T(u)$ and $T(v)$ whose weights are minimal. We introduce a new node w having u and v as children. This node w replaces u and v in the list of the children of s and with the same method as above we can encode $T(w)$ with $\max(weight(u), weight(v)) + 2$ colors.

Algorithm 1: Dichotomic encoding generation for a given tree T

Data : A tree T.
Result: A dichotomic encoding of T.
begin
  Each leaf v is given a weight: $weight(v) \leftarrow 0$;
  Other vertices are not given any weight;
  tant que *There exist a node s without weight such that all his clhildren have a weight* **faire**
    si *s has a unique child v* **alors**
      Give to v the color : $c(v) \leftarrow weight(v) + 1$;
      Give to s the weight : $weight(s) \leftarrow weight(v) + 1$;
    si *s has exactly 2 children u and v* **alors**
      Give to u the color : $c(u) \leftarrow \max(weight(u), weight(v) + 1$;
      Give to v the color : $c(v) \leftarrow \max(weight(u), weight(v) + 2$;
      Give to s the weight :
      $weight(s) \leftarrow \max(weight(u), weight(v)) + 2$;
    si *s has more than 3 children* **alors**
      Choose u and v two children whose weights are minimal;
      Introduce a new node w whose children are u and v, and which is the child of s;
      Give to u the color: $c(u) \leftarrow \max(weight(u), weight(v) + 1$;
      Give to v the color : $c(v) \leftarrow \max(weight(u), weight(v) + 2$;
      Give to w the weight :
      $weight(w) \leftarrow \max(weight(u), weight(v)) + 2$;
  Let T' be the tree generated from T, associate to each node s of T the following encoding : $code(s) = \{c(s') | s' \in T' \text{ and } s' \leq_{T'} s\}$;
end

The algorithm 1 follows this strategy. It computes a binary tree T' where the input tree T is embedded. At the end every node s of T' (the nodes of T and the new intermediate nodes) has a unique color stored in $c(s)$, it corresponds to the reduced encoding of our bit-vector encoding and the last step consists in propagating these colors through the hierarchy to obtain the final encoding.

Figure 6 shows an execution of Algorithm 1 which generates the tree T' from T. Intermediate nodes are in white, the color $c(s)$ of a node s is given by an integer, the integer given between parenthesis corresponds to the weight carried by the node. The last drawing represents the initial tree with its final bit-vector encoding (after step 1).

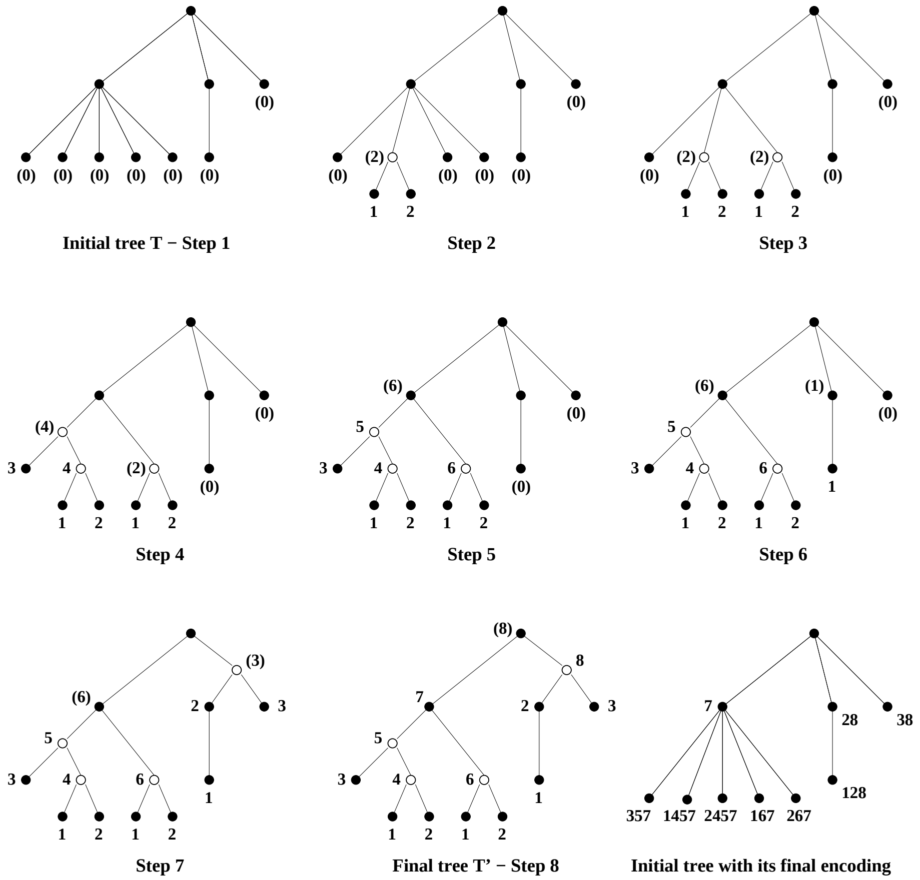


Fig. 6. Initial tree T, the binary tree T' and the final *dichotomic encoding* computed by algorithm 1

4 Results

First we will develop in this section the theorical aspects of our results. We will discuss about optimality and give some bounds for the size of the encoding computed by our algorithm. We will also study the case of two special shapes of tree hierarchies. Secondly we will sum up some pratical tests carried out on class hierarchies of oriented objects languages like Visualworks2 or Digitalk3.

4.1 Theoretical results

Optimality result. The next theorem garanties that in the framework of "splitting and balancing" strategies, our algorithm provides a bit-vector encoding of optimal size. Vitek and al. conjectured it was feasible in [9].

Theorem 1. *The size of the encoding generated by Algorithm 1 is the smallest for a dichotomic encoding.*

Proof. The proof shows that any optimal dichotomic encoding can be transformed into the dichotomic encoding generated by Algorithm 1 without changing its size. This proof is developed in the Appendix. □

Remark 1. the algorithm does not provide the optimal size of all the bit-vector encoding (as it will be seen on the generic examples of the next paragraph). As far as we know, finding polynomially this minimum size remains an open problem.

Evaluation for two particular instances of tree hierarchies. Our interest focusses on two generic examples illustrated by Figure 7 and Figure 8. A ***complete d-tree*** is a tree hierarchy such that all the nodes have a degree equal to d and all the chains from the root to the leaves have the same height (see Figure 7). A ***comb d-tree*** is a tree hierarchy such that all the nodes have a degree equal to d and the leaves are the children of a unique chain (see Figure 8). These generic examples give a good idea of the behaviour of known encoding algorithms. In Table 1, the encoding sizes of three algorithms are compared: Caseau's encoding [3], CHNR's encoding [18] (these two methods were mentionned in Section 2) and the encoding of algorithm 1. The method described in [9] does not appear in this table because it is not possible to predict the output size. We also give the number of nodes. In CHNR's column, the exact value is in fact obtained by replacing $\lceil \log_2(d) \rceil$ by $cmin(d)$ but we leave $\lceil \log_2(d) \rceil$ because it is approximately the real value. The parameter h is the height of the hierarchy.

Instance	Number of nodes	*Caseau's method*	*CHNR's method*	*Dichotomy*
Complete d-tree	d^h	$h * d$	$h * \lceil log_2 d \rceil$	$2 * h * \lceil log_2 d \rceil$
Comb d- tree	$h * d$	$h * d$	$h * \lceil log_2 d \rceil$	$2 * (h + \lceil log_2 d \rceil)$

Table 1. Comparison of different bit-vector methods

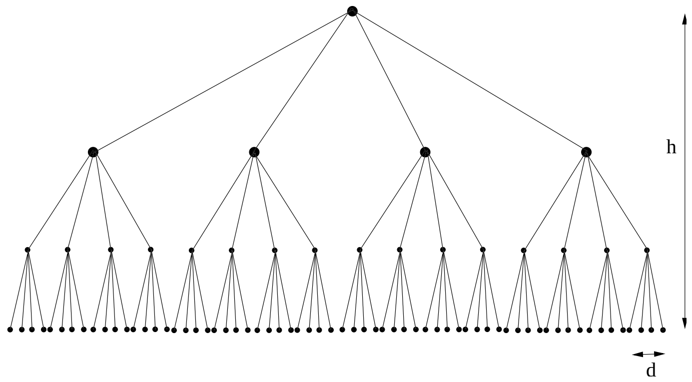


Fig. 7. Complete d-tree of height h with $d = 4$ and $h = 3$.

Some bounds on the encoding size. If we look at the examples of the table 1, our algorithm provides encodings with a size close to $\log_2(N)$ bits (where N is the number of nodes), especially when the height h is smaller than $\log_2(N)$. For these cases, the space cost is close to the encoding schemes which minimize the space storage and are not bit-vector encodings (such as the one presented in Section 2). The next proposition bounds the encoding size of our algorithm with usual paramaters of hierarchies. Let T be a tree hierarchy, the height of T is denoted by h and the maximal degree of a node of T is denoted by Δ. The number of nodes of T is N.

Proposition 1. *Let $size(T)$ be the size of the bit-vector encoding generated by Algorithm 1 for T. Then:*

$$\max(h, \lceil \log_2(N) \rceil) \leq size(T) \leq 2 * h * \lceil \log_2(\Delta) \rceil$$

On practical examples, we can expect the size to be far from the upper bound. We have experimented our algorithm on a set of benchmarks in the next subsection.

4.2 Practical results

In order to evaluate the performance of Algorithm 1, it has been applied to a collection of benchmarks. These benchmarks have been used by Vitek, Krall and Horspool in [9] and [16] for the analysis of several hierarchical encoding algorithms.

The benchmarks that we have used are four examples of hierarchies from object oriented languages like Smalltalk or C++. These class libraries contain from 311 to 1956 classes and have a tree structure.

Table 2 recapitulates the encoding size that were obtained for each of the four hierarchies: Visualworks3 (Smalltalk-80), Digitalk3 (Smalltalk-80), NeXTStep

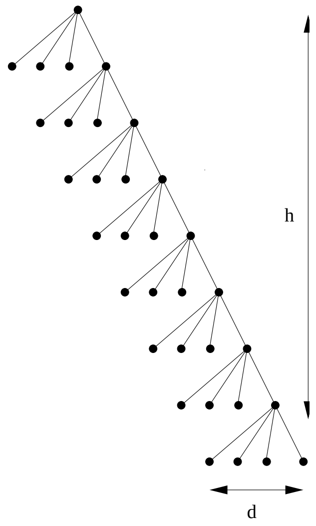


Fig. 8. Comb d-tree of height h with $d = 4$ and $h = 8$.

(Objective-C), ET++ (C++). The second column indicates the number of classes in the hierarchy. The next two columns have been picked up from the article [9]. The "Caseau" column gives the size of a bit-vector encoding obtained with the method described in [3] for tree hierarchies. The "KVH" column gives the best results obtained by the heuristics of Vitek and al. [9]. Their heuristics can use different preprocessing steps and different optimization routines. The results presented here correspond to the best choices in their strategy. We recall that their heuristics also apply to multiple subtyping hierarchies. The "CHNR" column presents the improvement of Caseau's method which was described in [18]. And the last column gives the size of our bit-vector encoding. Drawings of these hierarchies can be found in the Appendix, with some complements concerning the format of the benchmarks and our implementation.

Benchmark	Classes	*Caseau's method*	*KVH's method*	*CHNR's method*	*Dichotomy*
Visualworks2	1956	420	50	58	32
Digitalk3	1357	325	36	52	29
NeXTStep	311	177	23	28	20
ET++	371	181	30	41	20

Table 2. Encoding sizes for benchmarks

5 Incremental aspects

In this section, we suppose that we have encoded a tree hierarchy T with Algorithm 1 and that a new type x is added to this hierarchy. We also suppose that this addition does not introduce new subtype relations between the old types. We present here different ways to produce a bit-vector encoding of the new hierarchy. whether the addition maintain the tree structure and whether we want to keep on minimizing the encoding size.

If we have a sufficient amount of space, for instance if the encoding size is 40 bits and we have stored each bit-vector in two 32-bits words, we have spare bits available. We forget the optimization schemes and we can use the method mentioned in the introduction. This method works when x has at most one parent, it is the usual case for tree hierarchies. First the new type x gets the colors of its parent. Then we add a new color to the current set S (or equivalently we consider a new bit for all the bit-vectors) and we add this color to the code x and to the codes of all its descendants. Figure 9 illustrates this fast transformation (Color 7 has been added). Of course, after many additions, this method leads to an expensive bit-vector encoding. A possible strategy could consist in adding a new color for each new type and recomputing the whole encoding with an optimization algorithm as soon as the encoding size becomes too large.

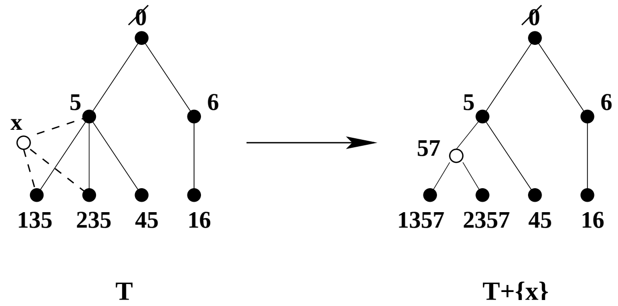


Fig. 9. A simple way to modify a bit-vector encoding after adding a node x

We have tested this method to encode hierarchies that are almost tree hierarchies. For a given hierarchy, suppose that we have a good knowledge of the hierarchy or an algorithm which enables us to find a large sub-hierarchy which is a tree. Then we can encode this tree hierarchy with Algorithm 1 and add one by one the other types of the initial hierarchy. The encoding size of such a method is the encoding size of the tree hierarchy plus the number of types that do not belong this tree hierarchy. For instance, two of the benchmarks of [9] are almost trees : the Unidraw library (614 C++ classes) and the Java API library from Sun (225 classes). The Unidraw library has 6 classes such that their removal yields a tree hierarchy. With Algorithm 1, we have obtained an encoding size for this tree hierarchy which is 22 bits, leading to a bit-vector encoding of the whole hierarchy of size 22+6=28 bits. The best value obtained in [9] for this hierarchy was 30 bits. The same method applied to the Java library gives a tree sub-hierarchy encoded with 18 bits and 6 classes out of this sub-hierarchy, leading to an encoding of 18+6=24 bits. The best value obtained in [9] for this hierarchy was 23 bits. The advantage of this method is that once the tree sub-hierarchy has been found, the computation of the bit-vector encoding is fast.

If we want to keep on minimizing the size of the encoding and if the addition of the new type preserves the tree structure, it is possible to compute a new dichotomic encoding of optimal size without applying Algorithm 1 to the whole hierarchy T. We need to keep the binary tree generated from T by Algorithm 1. Then we just have to recompute the weights of the nodes of the chain going from the new node x up to the root node. Two cases can occur. In the first case, x is introduced as a leaf in the tree hierarchy. Then we can recompute the encoding by giving to x a zero weight, removing all the weights of the chain going to the root node (its ancestors) and the corresponding intermediate nodes, and applying Algorithm 1 from the loop. The second case occurs when x has some children in the hierarchy. Then we can recompute the encoding by leaving x without any weight, removing all the weights of the chain going from x up to the root node and the corresponding intermediate nodes, and applying Algorithm 1 from the loop. In both cases, we obtain the optimal dichotomic encoding of the new hierarchy.

In all the other cases, if we want to keep on minimizing the encoding size, we can use the incremental algorithm proposed in [18]. This algorithm computes the conflict graph of the new hierarchy and then a coloration of this graph which leads to the encoding. The incremental aspect is that most of the vertices of this graph correspond to the old hierarchy and the preceding encoding enables to colour these vertices quickly and with few colors. Then we just have to colour efficiently the remaining vertices. This method is more general but it clearly takes much more time than the previous methods. We won't go into further details. For a complete description of this algorithm, see [18].

References

1. H. At-Kaci, R. Boyer, P. Lincoln, and R. Nasr. Efficient implementation of lattice operations. *ACM Transactions on Programming Langages and Systems*, 11(1):115–146, january 1989.
2. A. Bouchet. Codages et dimensions de relations binaires. *Annals of Discrete Mathematics 23, Ordres: Description and Roles, (M. Pouzet, D. Richard eds)*, 1984.
3. Yves Caseau. Efficient handling of multiple inheritance hierarchies. In *OOPSLA'93*, pages 271–287, 1993.
4. G. Ellis. Efficient retrieval from hierarchies of objects using lattice operations. In *Conceptual Graphs for knowledge representation, (Proc. International conference on Conceptual Structures, Quebec City, Canada, August 4-7, 1993), G. W. Mineau, B. Moulin and J. Sowa, Eds, Lecture Notes in Artificial Intelligence 699, Springer, Berlin*, 1993.

5. A. Fall. The foundations of taxomic encoding. in International KRUSE Symposium: Knowledge Retrieval, Use and Storage for Efficiency, University of Santa-Cruz., August 1995.
6. M. Habib, M. Huchard, and L. Nourine. Embedding partially ordered sets into chain-products. In *International KRUSE Symposium: Knowledge Retrieval, Use and Storage for Efficiency. University of Santa-Cruz.*, pages 147–161, August 1995.
7. M. Habib and L. Nourine. Bit-vector encoding for partially ordered sets. In Proc. of International Workshop ORDAL'94, editor, *Orders, Algorithms and Applications*, number 831, pages 1–12, Lyon, France, July 1994. LNCS.
8. M. Habib, L. Nourine, and O. Raynaud. A new lattice-based heuristic for taxonomy encoding. In *International KRUSE Symposium: Knowledge Retrieval, Use and Storage for Efficiency. Vancouver .*, pages 60–71, August 1997.
9. A. Krall, J. Vitek, and R.N. Horspool. Near optimal hierarchical encoding of types. In *Ecoop'97, LNCS*, number 1241, pages 128–145.
10. G. Markowsky. The representation of posets and lattices by sets. *Algebra Universalis*, 11:173–192, 1980.
11. N.H.Cohen. Type-extension type tests can be performed in constant time. *Programming languages and systems*, 13(4):626–629, July 1991.
12. O. Raynaud and E. Thierry. Calcul de la 2-dimension d'un ensemble ordonn. Research Report 00082, LIRMM, Montpellier, France, 2000.
13. M. Skorsky. *Endliche Verbnde Diagramme und Eigenschaften*. PhD thesis, (written in English) Darmstadt, Germany, 1992.
14. J. Stahl and R. Wille. Preconcepts of contexts. *in Proc. Universal Algebra (Sienna)*, 1984.
15. William T. Trotter. Embedding finite posets in cubes. *Discrete Mathematics*, 12:165–172, 1975.
16. J. Vitek, R.N. Horspool, and A. Krall. Efficient type inclusion tests. In *OOPSLA'97*, number 1241, pages 128–145.
17. R. Wille. Restructuring lattice theory: An approach based on hierarchies of contexts. *in Ordered sets,I. Rival, Eds. NATO ASI No 83, Reidel, Dordecht, Holland*, pages 445–470, 1982.
18. M. Habib Y. Caseau, L. Nourine, and O. Raynaud. Encoding of mutiple inheritance hierarchie and partial orders. *Computational Intelligence*, 15:50–62, N1, 1999.

Appendix

Analysis of benchmarks

In Section 4, we have compared the encoding sizes obtained with different heuristics on benchmarks extracted from the article of Vitek and al. [9]. These benchmarks are available from the address:

http://www.complang.tuwien.ac.at/andi/typecheck/

as well as the encoding program of Vitek and al. These are examples of class libraries from object oriented languages like Smalltalk or C++. We have applied our encoding algorithm to the ones that use single inheritance only. These four tree hierarchies contain from 311 to 1956 classes.

Our implementation of Algorithm 1 has been written in C++ (available from *http://www.lirmm.fr/ raynaud*). It takes as input a tree hierarchy under the format **.inh** - the format of the benchmarks. A hierarchy under this format is stored in a file where the nodes are given by their name and each ligne looks like **super class1 name class2**, which signifies that **class2** is a descendant of **class1**. Figure 10 shows an example of file with the associated hierarchy.

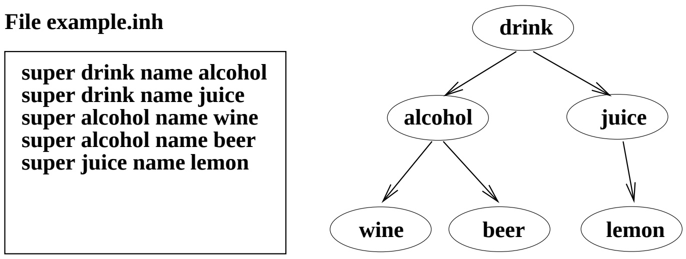


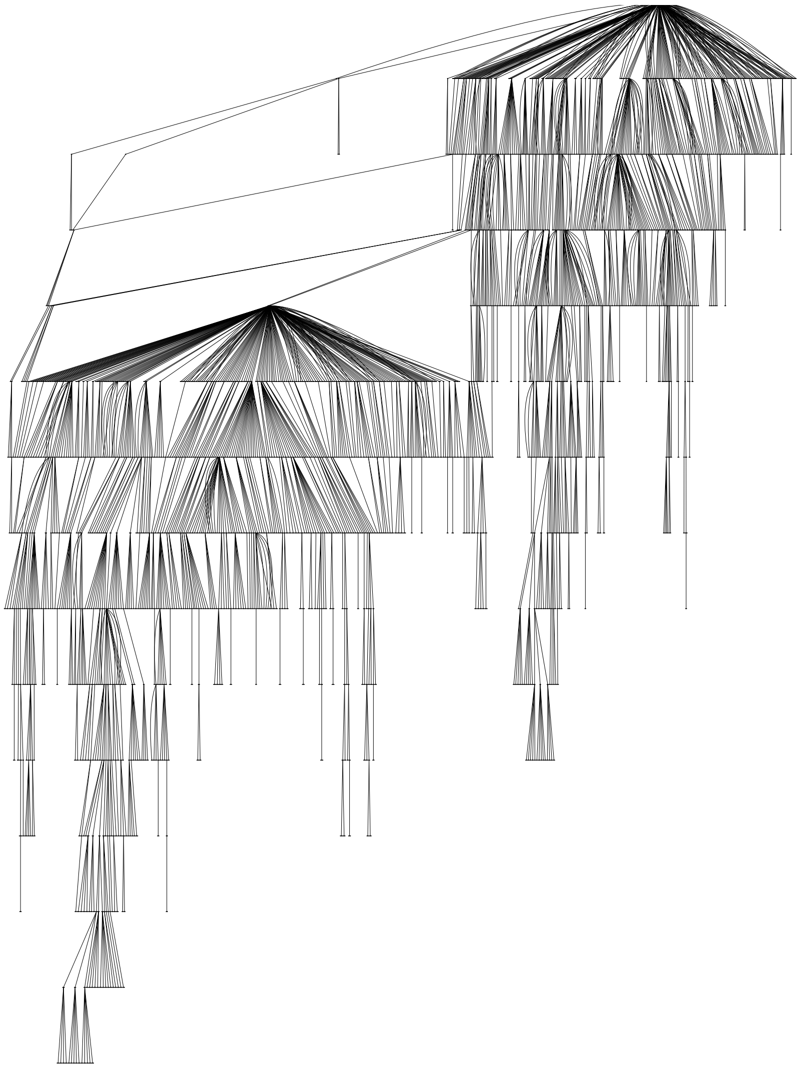
Fig. 10. Example of .inh syntax and the associated hierarchy.

The next pages present the drawings of the four tree hierarchies that have been used. These drawings have been generated by the freeware *Dotty* developed in *ATT*, which can be downloaded from:

http://www.research.att.com/sw/tools/graphviz/

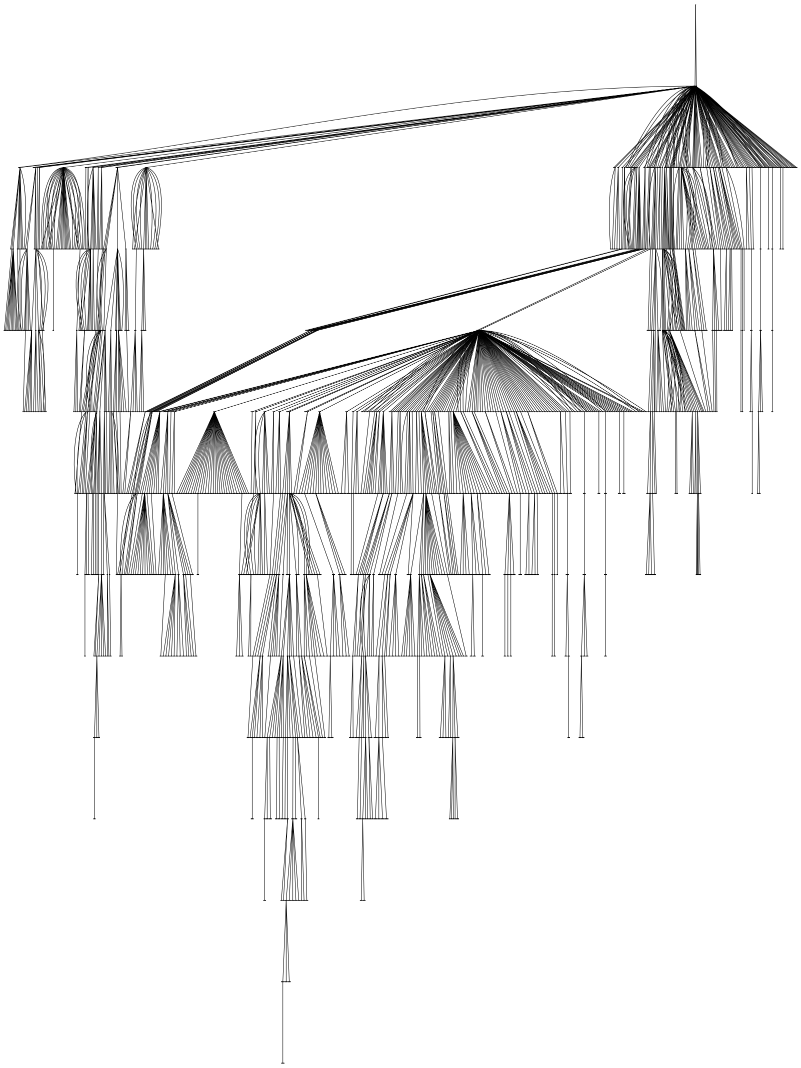
The size of the encoding produced by algorithm 1 lies at the bottom.

Visualworks2



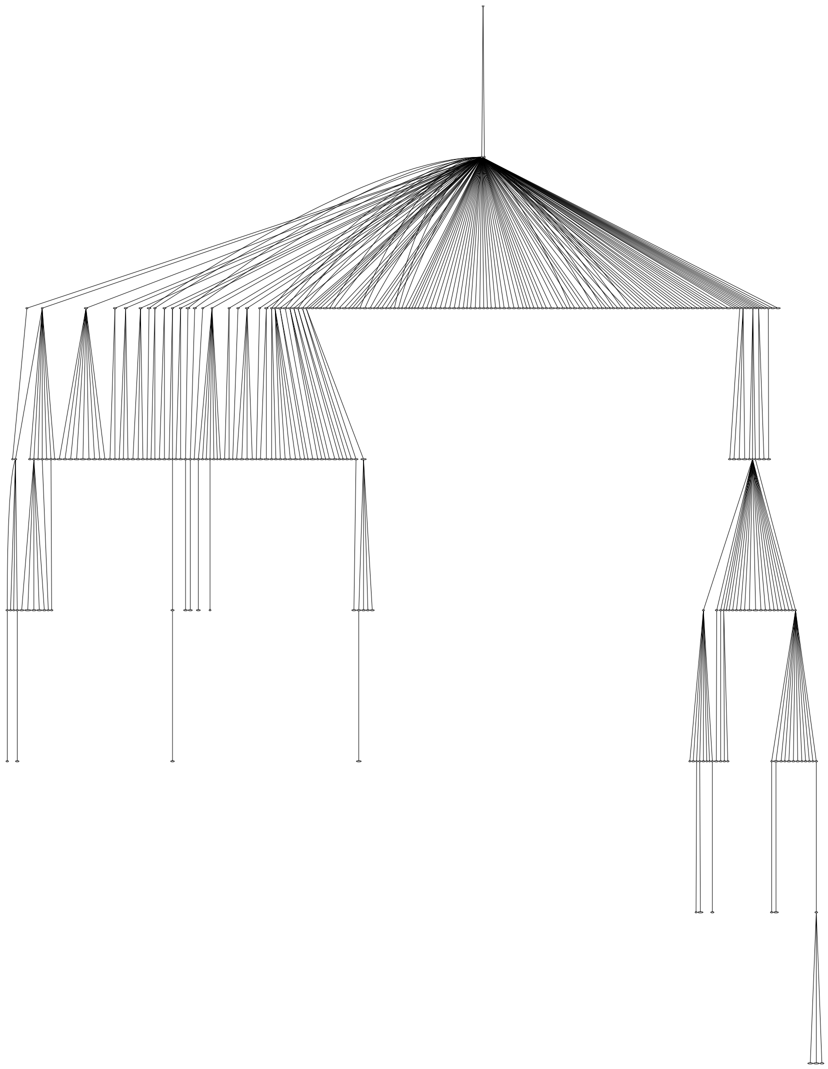
Programming language:	Smalltalk-80
Number of classes:	1956
Size of the bit-vector encoding:	32 (number of bits per class)

Digitalk3



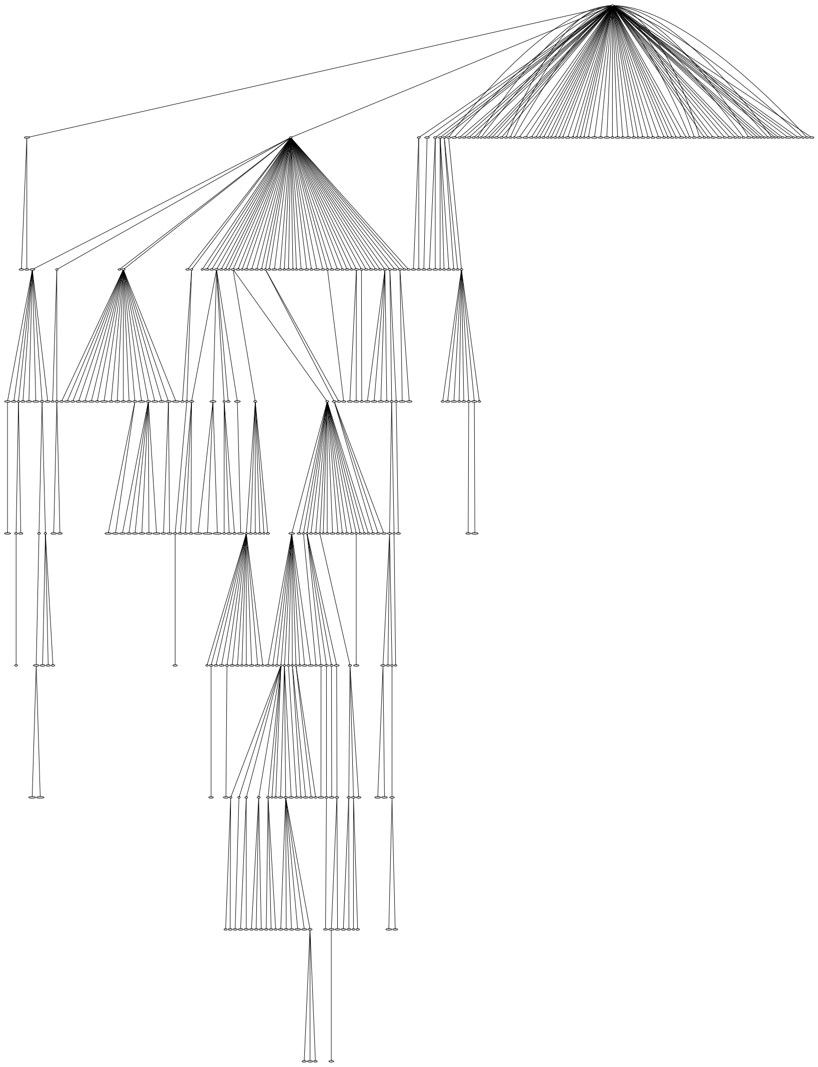
Programming language:	Smalltalk-80
Number of classes:	1357
Size of the bit-vector encoding:	29 (number of bits per class)

NeXTStep



Programming language:	Objective-C
Number of classes:	311
Size of the bit-vector encoding:	20 (number of bits per class)

ET++



Programming language:	C++
Number of classes:	371
Size of the bit-vector encoding:	20 (number of bits per class)

Proof of Theorem 1

We give here a sketch of the proof of Theorem 1 which is stated in Section 4.

Theorem 2. *The size of the encoding generated by Algorithm 1 is the smallest for a dichotomic encoding.*

Proof. Producing a dichotomic encoding of a tree T is equivalent to embedding T into a binary tree T' Then the size of the dichotomic encoding is equal to the maximal weight of a chain in the binary tree in the sense of Caseau (the weight of a chain is the sum of the degrees of the nodes of this chain). Moreover we can associate with each node s of T' a weight which is the maximal weight of a chain starting from s. This is equivalent to the number of colors encoding the subtree rooted at s in the dichotomic encoding.
To prove that our algorithm computes the smallest size for a dichotomic encoding, we just have to prove locally that it makes the best construction for a ***level***, namely a node with its weighted children.
Suppose that all the children of a node s carry a weight. We are going to prove that our construction minimizes the weight that will be propagated to the node s.

1. Consider an embedding of this level into a binary tree BT. We associate with each node x its height $h(x)$ in BT and each leaf x of BT carries a weight $w(x)$. The weight of BT, denoted by $weight(BT)$ is the maximum of $w(x) + 2 * h(x)$ for all the leaves of BT.
2. Suppose that the two leaves of BT with the smallest weights α and β are not "married" (the children of the same node). Then we can construct another binary tree BT' embedding the level such that α and β are married and $weight(BT') \leq weight(BT)$. Figure 11 describes this initial situation. The nodes carrying the weights α and β have a first common ancestor which is s', the weight α is "married" with γ and β is "married with δ. We suppose that $\alpha \leq \beta \leq \gamma \leq \delta$. The height of the parent of α and γ (resp. β and δ) from s' is denoted by h_γ (resp. h_δ). Now suppose that $h_\gamma \leq h_\delta$, by inverting δ and α we have a new binary tree BT' such that $weight(BT') \leq weight(BT)$ (if we had $h_\gamma \geq h_\delta$, we would have inverted γ and β).
3. Suppose now that we suppress two "married" leaves of BT with weights α and β and we associate with the new leaf (their common parent) the weight $\max(\alpha, \beta) + 2$. Then it is clear that this new binary tree BT' verifies $weight(BT') = weight(BT)$.

Given a level with weights on the leaves, consider an embedding BT of minimum weight $weight(BT)$. By alternatively using the transformations 2 and 3, we transform BT into another binary tree BT' with the same optimal weight and which is exactly the embedding constructed by Algorithm 1.

$\square$

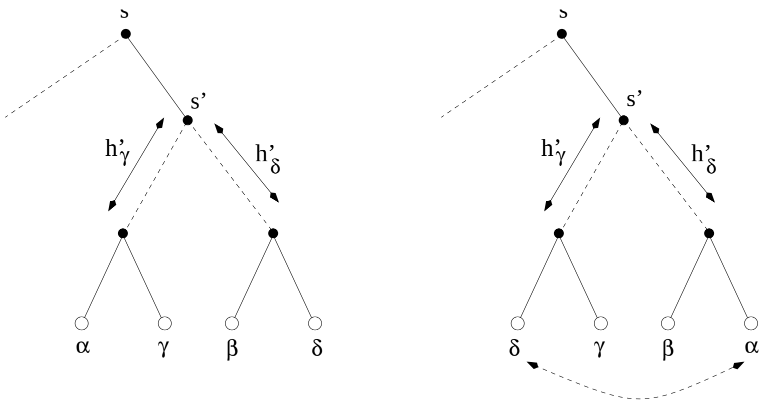


Fig. 11. Transformation of the embedding