Abstract. Any tree can be represented in a maximally compact form as a directed acyclic graph where common subtrees are factored and shared, being represented only once. Such a compaction can be effected in linear time. It is used to save storage in implementations of functional programming languages, as well as in symbolic manipulation and computer algebra systems. In compiling, the compaction problem is known as the “common subexpression problem” and it plays a central rôle in register allocation, code generation and optimisation. We establish here that, under a variety of probabilistic models, a tree of size $n$ has a compacted form of expected size asymptotically

$$C \frac{n}{\sqrt{\log n}},$$

where the constant $C$ is explicitly related to the type of trees to be compacted and to the statistical model reflecting tree usage. In particular the savings in storage approach 100% on average for large structures, which overperforms the commonly used form of sharing that is restricted to leaves (atoms).

Introduction. A tree can be compacted by representing occurrences of repeated subtrees only once. In that case, several pointers will point to the representation of any common subtree, and the original tree becomes a directed acyclic digraph also called a dag. The process itself is diversely known as “sharing” or “common subexpression recognition”. Obviously some storage is saved in this way, and our purpose is to estimate the expected gain attained by such a representation. (See Fig. 1.)

Trees that we consider are plane rooted trees [21] as commonly occur in a variety of contexts: in Lisp systems or as a representation of expressions in compiling; as syntax trees in parsing and code generation or structured programme editing; as the representation of terms in symbolic manipulation and computer algebra systems.

In the programming language Lisp [24], all programmes and data are represented in the form of “symbolic expressions” (S-expressions) having a binary branching tree structure. Under the tree representation, the external nodes are labelled with primitive symbols or atoms while the internal nodes, that are unlabelled, only reflect the hierarchical structure of the expression.

In most Lisp implementations, external nodes of (S-expression) trees are kept separately, in a storage area called “atom space”, and multiple instances of them are stored uniquely. This is a restricted form of sharing that presents obvious storage saving advantages. Following an original suggestion by McCarthy, this sharing scheme can also be extended to internal nodes. Shared representations of Lisp trees introduce greater complexity in the management
The syntax structure of a program is also described by a tree. Such a representation is invariably used in parsers and structured editors. For instance, the syntax oriented Mentor system \[12\] uses some amount of sharing to save space when storing the representation of a program on disk. In another context, common subexpression recognition leads to improved register allocation efficiency, a well known fact in compiler design \[1\]; this in turn results in code that is faster to execute. Alternatively, recognition of common program fragments is used to generate more compact compiled code by some compilers.

Unification itself, which is at the heart of logic programming systems is usually implemented using sharing in order to avoid the combinatorial explosion that would arise from repeated duplications of subtrees.

In a related context, a symbolic manipulation system like Maple \[8, 7\] manages storage using pointers and hashing in such a way that subexpressions exist uniquely in main memory. With this representation, the effect of applications of expansion rules that cause subtree duplications, like distributivity or symbolic differentiation, is somewhat decreased: For instance in \[18\], it is shown that symbolic differentiation has expected time and storage costs of \(O(n^{3/2})\) while the cost reduces to \(O(n)\) when sharing is used. Furthermore, in the Maple system, functions have a "remember" option (memo functions) and in conjunction with sharing, this feature appreciably improves time performances for many applications. This can be seen when computing, for instance

\[
\frac{d^3}{dx^3} \sin(x^4e^{\sqrt{1-x^4}}).
\]

There, a large number of duplicate computations take place unless some sharing is used. The Macsyma system also allows the user to save expressions in a shared format: this is the "Fas-save" command \[23, Chap. 10\].

All these applications demonstrate the interest of sharing subexpressions or subtrees in a diversity of contexts. It thus seems of interest to be able to quantify under well defined statistical models of tree usage the gains to be expected from tree compaction.

Each tree has a maximally factored representation as a dag which is unique up to isomorphism. The size of that dag representation, measured by the number of its nodes, will be referred to here as the \textit{compacted size} of the tree. The extremal cases for the cost of this structure are easy to characterize. In this paper, we consider trees with a fixed finite number of node types so that node degrees are bounded by some constant. Then, the \textit{compacted size of a tree with n nodes lies between } \(O(\log n)\) \text{ and } \(O(n)\). Our objective is to prove under a large