Software Clustering based on Information Loss Minimization

Periklis Andritsos
University of Toronto
periklis@cs.toronto.edu

Vassilios Tzerpos
York University
bil@cs.yorku.ca

The majority of the algorithms in the software clustering literature utilize structural information in order to decompose large software systems. Other approaches, such as using file names or ownership information, have also demonstrated merit. However, there is no intuitive way to combine information obtained from these two different types of techniques.

In this paper, we present an approach that combines structural and non-structural information in an integrated fashion. LIMBO is a scalable hierarchical clustering algorithm based on the minimization of information loss when clustering a software system.

We apply LIMBO to two large software systems in a number of experiments. The results indicate that this approach produces valid and useful clusterings of large software systems. LIMBO can also be used to evaluate the usefulness of various types of non-structural information to the software clustering process.

1 Introduction

It is widely believed that an effective decomposition of a large software system into smaller, more manageable subsystems can be of significant help to the process of understanding, redocumenting, or reverse engineering the system in question. As a result, the software clustering problem has attracted the attention of many researchers in the last two decades.

The majority of the software clustering approaches presented in the literature attempt to discover clusters by analyzing the dependencies between software artifacts, such as functions or source files [15, 12, 14, 21, 20, 8, 18, 11, 26]. Software engineering principles such as information hiding or high-cohesion, low-coupling are commonly employed to help determine the boundaries between clusters.

Other approaches have also demonstrated merit. Using naming information, such as file names or words extracted from comments in the source code [3, 16] may be the best way to cluster a given system. The ownership architecture of a software system, i.e. the mapping that shows which developer is responsible for what part of the system, can also provide valuable hints [5]. Some researchers have also attempted to combine structural information (based on dependencies) with non-structural one (based on naming) in their techniques [4]. Others have proposed ways of bringing clustering into a more general data management framework [1].

Even though the aforementioned approaches have shown that they can be quite effective when applied to large software systems, there are still several issues that can be identified:

1. There is no guarantee that the developers of a legacy software system have followed software engineering principles such as high-cohesion, low-coupling. As a result, the validity of the clusters discovered following such principles, as well as the overall contribution of the obtained decomposition to the reverse engineering process, can be challenged.

2. Software clustering approaches based on high-cohesion, low-coupling fail to discover utility subsystems, i.e. collections of utilities that do not necessarily depend on each other, but are used in many parts of the software system (they may or may not be omnipresent nodes [18]). Such subsystems do not exhibit high-cohesion, low-coupling, but they are frequently found in manually-created decompositions of large software systems.

3. It is not clear what types of non-structural information are appropriate for inclusion in a software clustering approach. Clustering based on the lines of code of each source file is probably inappropriate, but what about using timestamps? Ownership information has been manually shown to be valuable [6], but its effect in an automatic approach has not been evaluated.

In this paper, we present an approach that addresses these issues. Our approach is based on minimizing information loss during the software clustering process.

The objective of software clustering is to reduce the complexity of a large software system by replacing a set of objects with a cluster. Thus, the obtained decomposition is easier to understand. However, this process also reduces the amount of information conveyed by the clustered representation of the software system. Our approach attempts to create decompositions that convey as much information as possible by choosing clusters that represent their contents as accurately as possible. In other words, one can predict with high probability the features of a given object just by knowing the cluster that it belongs to.

Our approach clearly addresses the first issue raised above. It makes no assumptions about software engineering principles followed by the developers of the software system. It also creates decompositions that convey as much information about the software system as possible, a feature that should be helpful to the reverse engineer. Furthermore, as will be shown in Section 2, our approach can discover utility subsystems as well as ones based on high-cohesion, low-coupling. Finally, any type of non-structural information may be included in our approach. As a result, our approach can be used in order to evaluate the usefulness of various types of information such as timestamps or ownership. In fact, we present such a study in Section 5.

The structure of the rest of this paper is as follows: Section 2 presents some background from Information Theory, as well as the way our approach quantifies information loss for software systems. Section 3 presents LIMBO, a scalable hierarchical clustering algorithm based on the *Information Bottleneck* method [24]. In Section 4, we compare LIMBO to several other software clustering algorithms that have been presented in the literature. In Section 5, we use LIMBO in order to assess the usefulness of several types of non-structural information to the software clustering process. Finally, Section 6 concludes our paper.

2 Background

This section introduces the main concepts from Information Theory that will be used throughout the paper. We also give the formulation of the Information Bottleneck method and its use in software clustering.

2.1 Basics from Information Theory

In the following paragraphs we give some basic definitions of Information Theory and their intuition. These definitions can also be found in any information theory textbook, e.g. [9].

Throughout this section we will assume the dependency graph of an imaginary software system given in Figure 1. This

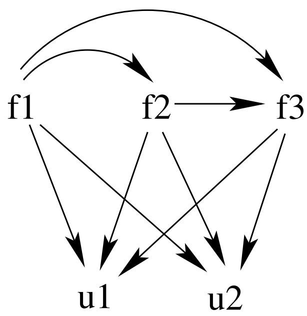

Figure 1. Example dependency graph

graph contains three program files f_1, f_2 and f_3 and two utility files u_1 and u_2. This software system is clearly too trivial to require clustering. However, it will serve as an example of how our approach discovers various types of subsystems.

Our approach starts by translating the dependencies shown in Figure 1 into the matrix shown in Table 1. The rows of this matrix represent the artifacts to be clustered while the columns represent the features that describe these artifacts. Since our example contains only structural information (non-structural information will be added in Section 2.4), the features of a software artifact are other artifacts. To avoid confusion, we will represent the software artifacts to be clustered with italic letters, *e.g.*, f_1, u_1, and the corresponding features with bold letters, *e.g.*, $\mathbf{f_1}, \mathbf{u_1}$.

In the matrix of Table 1, we indicate with 1, the presence of features and with 0 their absence. Note that, for a given artifact a, feature $\mathbf{f}$ is present if a depends on f, or f depends on a.

	$\mathbf{f_1}$	$\mathbf{f_2}$	$\mathbf{f_3}$	$\mathbf{u_1}$	$\mathbf{u_2}$
f_1	0	1	1	1	1
f_2	1	0	1	1	1
f_3	1	1	0	1	1
u_1	1	1	1	0	0
u_2	1	1	1	0	0

Table 1. Example matrix from dependencies in Figure 1

Let A denote a discrete random variable taking its values from a set $\mathbb{A}$. In our example, $\mathbb{A}$ is the set $\{f_1, f_2, f_3, u_1, u_2\}$. If $p(a)$ is the probability mass function of the values of A, the *entropy* $H(A)$ of variable A is defined by

$$H(A) = -\sum_{a \in \mathbb{A}} p(a) \log p(a)$$

Intuitively, entropy is a measure of disorder; the higher the entropy, the lower the certainty with which we can predict the value of A. We usually consider the logarithm with base two and thus entropy becomes the minimum number of bits required to describe variable A [9].

Now, let B be a second random variable taking values from the set $\mathbb{B}$ of all the features in the software system. In our example, $\mathbb{B}$ is the set $\{\mathbf{f_1}, \mathbf{f_2}, \mathbf{f_3}, \mathbf{u_1}, \mathbf{u_2}\}$. Then, $p(b|a)$ is the conditional probability of a value b of B given a value a of A. The *conditional entropy* $H(B|A)$ is defined as

$$\begin{aligned} H(B|A) &= \sum_{a \in \mathbb{A}} p(a) H(B|A=a) \\ &= -\sum_{a \in \mathbb{A}} p(a) \sum_{b \in \mathbb{B}} p(b|a) \log p(b|a) \end{aligned}$$

$H(B|A)$ gives the uncertainty with which we can predict the value of B given that a value of A appears.

An important question that arises is: "to what extent can the value of one variable be predicted from knowledge of the value of the other variable?". This question has a quantitative answer through the notion of *mutual information*, $I(A;B)$, which measures the amount of information that the variables hold about each other. The mutual information between two variables is the amount of uncertainty (entropy) in one variable that is removed by knowledge of the value of the other one. More precisely, we have

$$I(A;B) \quad = \quad H(A) - H(A|B) = H(B) - H(B|A)$$

Mutual information is symmetric, non-negative and equals zero if and only if A and B are independent.

Generally speaking, given a set of n values of A and a set of q values of B, we can conceptualize all features as an $n \times q$ matrix M, such as the one in Table 1, where each row holds the feature vector of an object. We normalize matrix M so that the entries of each row sum up to one. Hence, for an object a, the corresponding vector of the normalized matrix holds the conditional probability $p(B|A = a)$. The normalized matrix of Table 1 is depicted in Table 2.

$A \backslash B$	$\mathbf{f_1}$	$\mathbf{f_2}$	$\mathbf{f_3}$	$\mathbf{u_1}$	$\mathbf{u_2}$
f_1	0	1/4	1/4	1/4	1/4
f_2	1/4	0	1/4	1/4	1/4
f_3	1/4	1/4	0	1/4	1/4
u_1	1/3	1/3	1/3	0	0
u_2	1/3	1/3	1/3	0	0

Table 2. Normalized matrix of system features

Let us consider a particular clustering C_k of the elements of $\mathbb{A}$, so that every object $a \in \mathbb{A}$ is mapped to a cluster $c(a)$. Each of the clusters in C_k can be expressed as a vector over the features in $\mathbb{B}$, as explained in detail in Section 2.2.

We introduce a third random variable C taking values from set $\mathbb{C} = \{c_1, c_2, ..., c_k\}$, where $c_1, c_2, ..., c_k$ are the k clusters of C_k. The mutual information $I(B;C)$ quantifies the information about the values of B (the features of the software system) provided by the identity of a cluster (a given value of C). The higher this quantity is the more informative the cluster identity is about the features of its constituents. Therefore, our goal is to choose C_k in such a way that it maximizes the value of $I(B;C)$.

The maximum value for $I(B;C)$ occurs when $|\mathbb{C}| = |\mathbb{A}|$, *i.e.* each cluster contains only one object. The minimum value for $I(B;C)$ occurs when $|\mathbb{C}| = 1$, *i.e.* when all objects are clustered together. Interesting are the cases in-between, where we seek a *k-clustering* C_k, that contains a sufficiently small number of clusters (compared to the number of objects), while retaining a high value for $I(B;C)$.

Tishby et al., [24], proposed a solution to this optimization problem in what is termed the *Information Bottleneck Method*. Finding the optimal clustering is an NP-complete problem [10]. The next section presents a heuristic solution to maximizing $I(B;C)$.

2.2 Agglomerative Information Bottleneck

Slonim and Tishby [22] propose a greedy agglomerative approach, the *Agglomerative Information Bottleneck (AIB)* algorithm, for finding an informative clustering. This technique has also been used in document clustering [23] and the classification of galaxy spectra [19]. Similar to all agglomerative (or bottom-up) techniques, the algorithm starts with the clustering C_n, in which each object $a \in \mathbb{A}$ is a cluster by itself. As stated before, $I(A;B) = I(C_n;B)$. At step $n - \ell + 1$ of the AIB algorithm, two clusters c_i, c_j in ℓ-clustering C_ℓ are merged into a single component c^* to produce a new $(\ell - 1)$-clustering $C_{\ell-1}$. As the algorithm forms clusterings of smaller size, the information that the clustering contains about the features in $\mathbb{B}$ decreases; that is, $I(B;C_{\ell-1}) \leq I(B;C_\ell)$. The clusters c_i and c_j to be merged are chosen to minimize the information loss in moving from clustering C_ℓ to clustering $C_{\ell-1}$. This information loss is given by $\delta I(c_i, c_j) = I(B;C_\ell) - I(B;C_{\ell-1})$. We can also view the information loss as the increase in the uncertainty of predicting the features in the clusters before and after the merge.

After merging clusters c_i and c_j, the new component $c^* = c_i \cup c_j$ has, [22]

$$p(c^*|a) = \begin{cases} 1 & \text{if } a \in c_i \text{ or } a \in c_j \\ 0 & otherwise \end{cases} \tag{1}$$

$$p(c^*) = p(c_i) + p(c_j) \tag{2}$$

$$p(b|c^*) = \frac{p(c_i)}{p(c^*)}p(b|c_i) + \frac{p(c_j)}{p(c^*)}p(b|c_j) \tag{3}$$

Tishby et al. [24] show that

$$\delta I(c_i, c_j) = [p(c_i) + p(c_j)] \cdot D_{JS}[p(b|c_i), p(b|c_j)]$$

where D_{JS} is the *Jensen-Shannon (JS)* divergence, defined as follows. Let $p_i = p(b|c_i)$ and $p_j = p(b|c_j)$ and let

$$\bar{p} = \frac{p(c_i)}{p(c^*)}p_i + \frac{p(c_j)}{p(c^*)}p_j$$

denote the weighted average distribution of distributions p_i and p_j. Then, the D_{JS} distance is:

$$D_{JS}[p_i, p_j] = \frac{p(c_i)}{p(c^*)}D_{KL}[p_i||\bar{p}] + \frac{p(c_j)}{p(c^*)}D_{KL}[p_j||\bar{p}] \; .$$

D_{KL} is the *Relative Entropy*, or the *Kullback-Leibler (KL)* divergence, a standard information-theoretic measure of the difference between two probability distributions. Given two distributions p and q over a set A, the relative entropy is

$$D_{KL}[p||q] = \sum_{a \in \mathbb{A}} p(a) \log \frac{p(a)}{q(a)} \; .$$

Intuitively, the relative entropy $D_{KL}[p||q]$ is a measure of the redundancy in an encoding that assumes the distribution q, when the true distribution is p.

Then, D_{JS} distance is the average D_{KL} distance of p_i and p_j from $\bar{p}$. It is non-negative and equals zero if and only if $p_i = p_j$. It is also bounded above by one, and it is symmetric. Note that the information loss for merging clusters c_i and c_j, $\delta I(c_i, c_j)$, depends only on the clusters c_i and c_j, and not on other parts of the clusterings C_ℓ and $C_{\ell-1}$.

Intuitively, at each step, AIB merges two clusters that will incur the smallest value in δI. The probability of the newly formed cluster becomes equal to the sum of probabilities of the two clusters (equation (2)) and the conditional probability of the features given the identity of the new cluster is a weighted average of the conditional probabilities in the clusters before the merge (equation (3)).

Recasting the problem of software clustering within the context of the Information Bottleneck method, we consider as input an $n \times q$ table similar to the one in Table 2. For each vector a_i, which is expressed over q_i non-zero features we define

$$p(a_i) = 1/n \qquad (4)$$

$$p(b|a_i) = \begin{cases} 1/q_i & \text{if } M[a_i, b] \text{ is non-zero} \\ 0 & \text{otherwise} \end{cases} \qquad (5)$$

We can now compute the mutual information $I(A; B)$ and proceed with the Information Bottleneck method to cluster the vectors of the values in A.

2.3 Structural Example

By applying the equations of the previous section to the example software system presented in Section 2.1, we can compute all pairwise values of information loss (δI). These values are given in Table 3. The value in position (i, j) indicates the information loss we would incur, if we chose to group the i-th and the j-th artifact together.

	f_1	f_2	f_3	u_1	u_2
f_1	-	0.10	0.10	0.17	0.17
f_2	0.10	-	0.10	0.17	0.17
f_3	0.10	0.10	-	0.17	0.17
u_1	0.17	0.17	0.17	-	**0.00**
u_2	0.17	0.17	0.17	**0.00**	-

Table 3. Pairwise δI values for vectors of Table 2

Clearly, if utility files u_1 and u_2 get merged in the same cluster, c_u, we lose no information about the system, something that agrees with our intuition just by observation of Figure 1, which suggests that u_1 and u_2 have exactly the same structural features. On the other hand, we lose some information if f_1 and f_2 get merged in the same cluster c_f, which is the same loss of information if any pair among the program files forms a cluster. Table 4 depicts the new matrix after forming clusters c_f and c_u. Intuitively, c_u represents the dependencies of its constituents *exactly* as good as u_1 and u_2 before the merge, while c_f is *almost* as good. We compute the probabilities of the two new clusters using equation (2) from Section 2.2 as $p(c_f) = 2/5$ and $p(c_u) = 2/5$, while the new distributions $p(B|c_f)$ and $p(B|c_u)$ are calculated using equation (3) of the same section. The obtained values are shown in Table 4.

$A \backslash B$	$\mathbf{f_1}$	$\mathbf{f_2}$	$\mathbf{f_3}$	$\mathbf{u_1}$	$\mathbf{u_2}$
c_f	1/8	1/8	1/4	1/4	1/4
f_3	1/4	1/4	0	1/4	1/4
c_u	1/3	1/3	1/3	0	0

Table 4. Normalized matrix after forming c_f and c_u

The new matrix of pairwise distances is given in Table 5, which suggests that c_f will next be merged with f_3 as their δI value is the minimum. This indicates that our approach is able to discover both utility subsystems (such as c_u) as well as cohesive ones (such as the cluster containing f_1, f_2, and f_3).

	c_f	f_3	c_u
c_f	-	**0.04**	0.26
f_3	**0.04**	-	0.24
c_u	0.26	0.24	-

Table 5. Pairwise δI after forming c_f and c_u

2.4 Non-Structural Example

One of the strengths of our approach is its ability to consider various types of information about the software system. Our example so far contained only structural data. We will now expand it to include non-structural data as well, such as the name of the developer, or the location of an artifact.

All we need to do is extend the universe $\mathbb{B}$ to include the values of non-structural features. Of course, q_i will now be the number of both structural and non-structural features over which each vector a_i is expressed. This way our algorithm is able to cluster the software system in the presence of meta-information about software artifacts.

The files of Figure 1 together with their *developer* and *location* are given in Table 6.

The normalized matrix when $\mathbb{B}$ is extended to $\{\mathbf{f_1}, \mathbf{f_2}, \mathbf{f_3}, \mathbf{u_1}, \mathbf{u_2}, \mathbf{Alice}, \mathbf{Bob}, \mathbf{p_1}, \mathbf{p_2}, \mathbf{p_3}\}$ is given in Table 7.

After that, $I(A; B)$ is defined and clustering can be performed as in the case of structural data, without necessarily

	$\mathbf{f_1}$	$\mathbf{f_2}$	$\mathbf{f_3}$	$\mathbf{u_1}$	$\mathbf{u_2}$	**Alice**	**Bob**	$\mathbf{p_1}$	$\mathbf{p_2}$	$\mathbf{p_3}$
f_1	0	1/6	1/6	1/6	1/6	1/6	0	1/6	0	0
f_2	1/6	0	1/6	1/6	1/6	0	1/6	0	1/6	0
f_3	1/6	1/6	0	1/6	1/6	0	1/6	0	1/6	0
u_1	1/5	1/5	1/5	0	0	1/5	0	0	0	1/5
u_2	1/5	1/5	1/5	0	0	1/5	0	0	0	1/5

Table 7. Normalized matrix of system dependencies with structural and non-structural features

	Developer	Location
f_1	**Alice**	$\mathbf{p_1}$
f_2	**Bob**	$\mathbf{p_2}$
f_3	**Bob**	$\mathbf{p_2}$
u_1	**Alice**	$\mathbf{p_3}$
u_2	**Alice**	$\mathbf{p_3}$

Table 6. Non-structural features for the files in Figure 1

giving the same results. More on this issue will be presented in the experimental evaluation section of this paper.

3 Clustering using *LIMBO*

Given a large number of vectors n, the Agglomerative Information Bottleneck algorithm suffers from high computational complexity, namely $\mathcal{O}(n^2 \log n)$, which is prohibitive for large data sets. In this section we introduce the *scaLable InforMation BOttleneck (LIMBO)* algorithm that uses distributional summaries in order to deal with large data sets. We employ an approach similar to the one used in the BIRCH clustering algorithm for clustering numerical data [28]. However our distance measure is based on the IB method and we consider a different definition of summaries.

3.1 Distributional Cluster Features

We now introduce the notion of *Distributional Cluster Feature (DCF)*. Each cluster of vectors that LIMBO creates has a corresponding DCF. The DCF provides a summary of the cluster which is sufficient for computing the distance between two clusters or between a cluster and a single vector.

Let A, B, C, be random variables, and $\mathbb{A}$, $\mathbb{B}$, $\mathbb{C}$, be sets as defined in Section 2. The *Distributional Cluster Feature* (*DCF*) of a cluster $c \in \mathbb{C}$ is defined by the pair

$$DCF(c) = \Big(n(c), p(B|c)\Big)$$

where $n(c)$ is the number of vectors merged in c and $p(B|c)$ is the conditional probability distribution of the features in cluster c. In the case that c consists of a single vector a_i, $n(c) = 1$ and the values of $p(B|c)$ can be computed using equation 5.

For larger clusters, the DCF is computed recursively as follows. Let c^* denote the cluster we obtain by merging two clusters c_1 and c_2. The DCF of the cluster c^* is

$$DCF(c^*) = \Big(n(c_1) + n(c_2), p(B|c^*)\Big) \tag{6}$$

where

$$p(B|c^*) = \frac{n(c_1)}{n(c_1) + n(c_2)} p(B|c_1) + \frac{n(c_2)}{n(c_1) + n(c_2)} p(B|c_2)$$

as suggested by the expressions of the IB method.

The distance between two clusters c_1 and c_2 (denoted by $d(c_1, c_2)$) is the information loss incurred by their merge and is given by the expression $\delta I(c_1, c_2) = I(A; C_b) - I(A; C_a)$, where C_b and C_a denote the clusterings before and after the merge, respectively. Note that the information loss depends only on the clusters c_1 and c_2, and not on other parts of the clustering C_b. From the expressions in Section 2.2, we have that

$$d(c_1, c_2) = \left(\frac{n(c_1)}{n} + \frac{n(c_2)}{n}\right) D_{JS}[p(B|c_1), p(B|c_2)]$$

where n is the total number of vectors in the data set.

The DCFs can be stored and updated incrementally. The probability vectors are stored as sparse vectors, reducing the amount of space considerably.

3.2 The LIMBO clustering algorithm

We now present the LIMBO algorithm. In what follows, n is the number of input vectors, q is the number of features, and k is the chosen number of clusters. The LIMBO algorithm proceeds in four phases. In the first phase, we construct a DCF tree that summarizes the data. In the second phase, the DCFs of the tree are merged to produce a chosen number of clusters. In the third phase, we *label* the data, *i.e.* we associate each vector with the DCF to which the vector is closest. Finally, phase 4 determines an appropriate number of clusters for the proposed decomposition.

Phase 1: Creation of the DCF tree. As the name implies, the DCF tree is a tree whose nodes contain DCFs. The number of DCFs in each node is a parameter called *branching factor* (we will denote it by E). Figure 2 presents a DCF tree with a branching factor of 6.

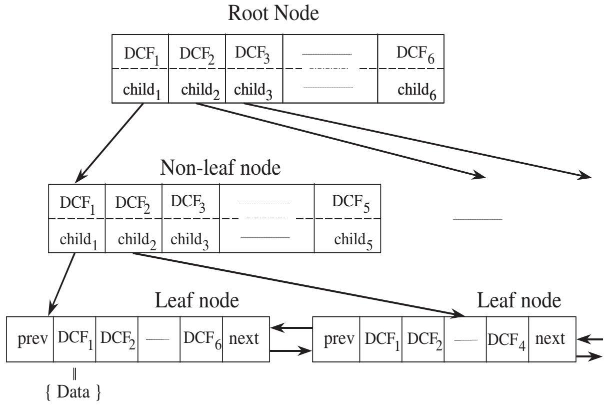

Figure 2. A DCF Tree with branching factor 6.

In order to create the DCF tree, we start with an empty root node. Vectors are processed one by one. A vector a is converted into $DCF(a)$, as described in Section 3.1. Then, starting at the root, we trace a path downward in the DCF tree as follows:

When at a non-leaf node, we compute the distance between $DCF(a)$ and each DCF entry of the node (a maximum of E entries), finding the closest DCF entry to $DCF(a)$. We follow the child pointer of this entry to the next level of the tree.

When at a leaf node, let $DCF(c)$ denote the DCF entry in the leaf node that is closest to $DCF(a)$. $DCF(c)$ is the summary of some cluster c. At this point we need to decide whether the vector $DCF(a)$ will be absorbed in $DCF(c)$ or not. If the distance $d(c,a)$, that is, the information loss incurred by merging $DCF(a)$ into $DCF(c)$ is less than a chosen threshold (discussed in detail in Section 3.3), then we proceed with the merge. Otherwise, a forms a cluster of its own. In this case, if there is space for another entry in the leaf node, $DCF(a)$ is inserted and all DCFs in the path toward the root are updated using Equation (6). If there is no space, the leaf node has to be split into two leaves. This is done in a manner similar to that in BIRCH [28]. We find the two DCFs in the node that are farthest apart and we use them as seeds for the new leaves. The remaining DCFs and $DCF(a)$ are placed in the leaf that contains the seed DCF to which they are closest.

When a leaf node is split, resulting in the creation of a new leaf node, its parent is updated, and a new entry is created at the parent node that describes the newly created leaf. If there is space in the parent node, we add a new DCF entry, otherwise the parent node must also be split. This process continues upward in the tree until the root is either updated or split itself. In the latter case, the height of the tree is increased by one.

The computational complexity of this phase is $\mathcal{O}(qEn\log_E n)$ since we need to visit E entries of each node and traverse a tree of height $n\log_E n$. Each computation of a distance requires the traversal of q entries in the worst case. The I/O cost of the algorithm is $\mathcal{O}(n)$ since only one scan of the data is required.

Phase 2: Clustering. After the construction of the DCF tree, the leaf nodes hold the DCFs of a clustering of the vectors in A. In this phase, our algorithm employs the *Agglomerative Information Bottleneck (AIB)* algorithm to cluster the DCFs in the leaves and produce a clustering C of the DCFs. The input to the AIB algorithm is the set of the conditional probability distributions $p(B|c*)$ stored in the leaf DCFs. The time for this phase depends upon the number of leaf DCFs (denoted by L).

In the worst case, the computational complexity of this phase is $\mathcal{O}(L^2)$, since all pairwise distances of the leaf entries need to be known in advance. There is no I/O cost involved in this phase since all computations are done in main memory.

Phase 3: Associating objects with clusters. Given a number of clusters k, Phase 2 produces k DCFs that serve as *representatives* of k clusters. In the third phase, we perform a scan over the data set and assign each vector to the cluster whose representative is closest with respect to the D_{KL} distance.

The I/O cost of this phase is the reading of the data set from the disk again. The CPU complexity is $\mathcal{O}(kqn)$, since each vector is compared against the k DCFs that represent the clusters.

Phase 4: Determining the number of clusters.

At this point, LIMBO is ready to produce a decomposition that exhibits small information loss for any k. In order to choose an appropriate number of clusters, we start by creating decompositions for all values of k between 2 and a large value. For the experiments performed for this paper, the chosen value was 100. We felt that clusterings of higher cardinality would not be useful from a reverse engineering point of view. Moreover, this value was always sufficient for the purposes of choosing an appropriate k.

Let C_k be a clustering of k clusters and C_{k+1} a clustering of $k+1$ clusters. If the cluster representatives created in Phase 2 reflect inherent groupings in the data, then these neighbouring clusterings must differ in only one cluster. More precisely, if two of the clusters in C_{k+1} get merged, this should result in C_k. Using MoJo [27], we can detect these clusterings by computing the distance between C_{k+1} and C_k, *i.e.* the value of $MoJo(C_{k+1}, C_k)$. If this value is equal to one, the difference between the two clusterings is a single join of two clusters of C_{k+1}, to produce the k clusters of C_k. As a result, k is chosen as the smallest value, for which $MoJo(C_{k+1}, C_k) = 1$.

3.3 Threshold value

LIMBO uses a threshold value to control the decision to merge a new vector into an existing cluster, or place it in a cluster by itself. This threshold limits the amount of information loss in our summary of the data set. It also affects the size of the DCF tree which in turn determines the computational cost of the AIB algorithm in Phase 2.

The threshold value $\tau(\phi)$, which is a function of a user-specified parameter ϕ, controls the decision between merging a new vector into an existing cluster, or placing it in a cluster by itself. Therefore, ϕ (and $\tau(\phi)$ in turn) affect the size of the DCF tree, as well as the granularity of the resulting representation. The value of $\tau(\phi)$ also controls the duration of Phase 2 of LIMBO's execution. A good choice for ϕ is necessary to produce a concise and useful summarization of the data set.

In LIMBO, we adopt a heuristic for setting the value $\tau(\phi)$ that is based on the mutual information between variables A and B. Before running LIMBO, the value of $I(A;B)$ is calculated by doing a full scan of the data set. Since there are n vectors in A, "on average" every vector contributes $I(A;B)/n$ to the mutual information $I(A;B)$. We define the threshold $\tau(\phi)$ as follows:

$$\tau(\phi) = \phi \frac{I(A;B)}{n} \tag{7}$$

where $0 \leq \phi \leq n$, denotes the multiple of the "average" mutual information that we wish to preserve when merging a vector into a cluster. If the merge would incur information loss more than ϕ times the "average" mutual information, then the new vector is placed in a cluster by itself.

In the next section we present our experiments in the software domain for $\phi = 0.0$. Given the relatively small size of the input it was not deemed necessary to consider higher values of ϕ. We also set $E = 4$, so that the Phase 1 insertion time is manageable (smaller values of E lead to higher insertion cost due to the increased height of the DCF tree). Experiments in different domains can be found in [2].

4 Experiments with Structural Input

In order to evaluate the applicability of LIMBO to the software clustering problem, we applied it to two large software systems of known authoritative decomposition, and compared its output to that of other well-established software clustering algorithms.

The two large software systems we used for our experiments were of comparable size, but of different development philosophy:

1. **TOBEY**. This is a proprietary industrial system that is under continuous development. It serves as the optimizing back end for a number of IBM compiler products. The version we worked with was comprised of 939 source files and approximately 250,000 lines of code. The authoritative decomposition of TOBEY was obtained over a series of interviews with its developers.

2. **Linux**. We experimented with version 2.0.27a of this free operating system that is probably the most famous open-source system. This version had 955 source files and approximately 750,000 lines of code. The authoritative decomposition of Linux was presented in [7].

The software clustering approaches we compared LIMBO to, were the following:

1. **ACDC**. This is a pattern-based software clustering algorithm that attempts to recover subsystems commonly found in manually-created decompositions of large software systems [26].

2. **Bunch**. This is a suite of algorithms that attempt to find a decomposition that optimizes a quality measure based on high-cohesion, low-coupling. We experimented with two versions of a hill-climbing algorithm, we will refer to as NAHC and SAHC (for nearest- and shortest-ascend hill-climbing) [15].

3. **Cluster Analysis Algorithms**. We also compared LIMBO to several hierarchical agglomerative cluster analysis algorithms. We used the Jaccard co-efficient that has been shown to work best in a software clustering context [4]. We experimented with four different algorithms: single linkage (SL), complete linkage (CL), weighted average linkage (WA), and unweighted average linkage (UA).

In order to compare the output of the algorithms to the authoritative decomposition, we used the MoJo distance measure[1] [25, 27]. Intuitively, the smaller the distance of a proposed decomposition to the authoritative one, the more effective the algorithm that produced it. Other comparison techniques, such as the ones introduced by Koschke and Eisenbarth [13], and Mitchell and Mancoridis [17] could also have been chosen. Part of our future work will include determining whether the choice of comparison technique affects the obtained results in a significant fashion.

For the experiments presented in this section, all algorithms were provided with the same input, the dependencies between the software artifacts to be clustered. The traditional cluster analysis algorithms were run with a variety of cut-point heights. The smallest MoJo distance obtained is reported below. This biases the results in favour of the cluster analysis algorithms, since in a different setting the cut-point height would have to be estimated without knowledge of the authoritative decomposition. However, as will be shown shortly, LIMBO outperforms the cluster analysis algorithms despite this bias.

Table 8 presents the results of our experiments. As can be seen, LIMBO created a decomposition that is closer to the authoritative one for both TOBEY and Linux, although the nearest-ascend hill-climbing algorithm of Bunch comes very

[1] A Java implementation of MoJo is available for download at: `http://www.cs.yorku.ca/~bil/downloads`

	TOBEY	Linux
LIMBO	311	237
ACDC	320	342
NAHC	382	249
SAHC	482	353
SL	688	402
CL	361	304
WA	351	309
UA	354	316

Table 8. MoJo distances between decompositions proposed by eight different algorithms and the authoritative decompositions for TOBEY and Linux

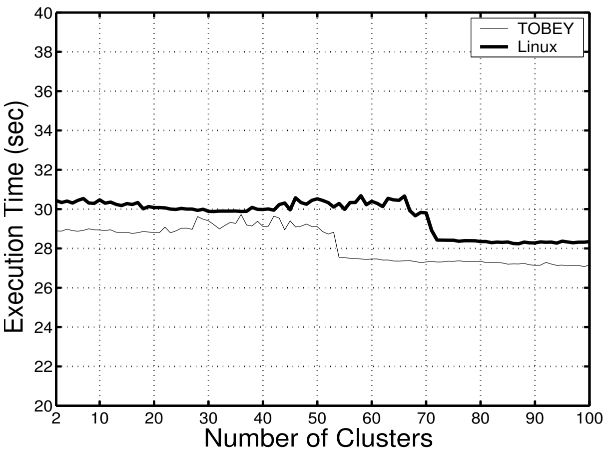

Figure 3. LIMBO execution time

close in the case of Linux, as is ACDC in the case of TOBEY. The cluster analysis algorithms perform respectably, but as can be expected, cannot be as effective as the specialized software clustering algorithms.

We believe that the fact that LIMBO performed better than other algorithms can be attributed mostly to its ability to discover utility subsystems. An inspection of the authoritative decompositions for TOBEY and Linux revealed that they both contain such collections of utilities. Since in our experience that is a common occurrence, we are optimistic that similar results can be obtained for other software systems as well.

The results of these experiments indicate that the idea of using information loss minimization as a basis for software clustering has definite merit. Even though further experimentation is required in order to assess the usefulness of LIMBO to the reverse engineering process, it is clear that it can create decompositions that are close to the ones prepared by humans.

We also tested LIMBO's efficiency with both systems. The time required to cluster a software system depends on the number of clusters determined in Phase 4. For a given number of clusters k, LIMBO was able to produce a C_k clustering within 31 seconds. Figure 3 presents LIMBO's execution time for both example systems, and all values of k from 2 to 100.

As can be seen on Figure 3, execution time varies only slightly as k increases. As a result, obtaining an appropriate clustering for either example system was a matter of minutes. The similarity in the efficiency of LIMBO for the two systems does not come as a surprise, since the number of source files to be clustered was similar (939 in TOBEY and 955 in Linux).

In the next section, we utilize LIMBO's ability to combine structural and non-structural information seamlessly in order to evaluate the usefulness of certain types of information to the reverse engineering process.

5 Experiments with Non-Structural Input

We now present results for the application of LIMBO to Linux when non-structural features are present. We will test the quality of clustering when the following features are added to the structural information:

- *Developers (dev)*: This feature gives the ownership information, *i.e.*, the names of the developers involved in the implementation of the file. In case no developer was known, we used a unique dummy value for each file.
- *Directory Path (dir)*: In this feature we include the full directory path for each file. In order to increase the similarity of the files residing in similar directory paths, we include the set of all subpaths for each path. For example, the directory information for file `drivers/char/ftape/ftape-io.c` is the set {`drivers`, `drivers/char`, `drivers/char/ftape`} of directory paths.
- *Lines of Code (loc)*: This feature includes the number of lines of code for each of the files. We discretized the values using two different schemes.
  1. The first scheme divides the full range of *loc* values into the intervals $(0, 100], (100, 200], (200, 300]$ etc. Each file is given a feature such as `RANGE1, RANGE2, RANGE3` etc.
  2. The second scheme divides the full range of *loc* values so that each interval contains the same number of values. Files are given features in a similar manner to the previous scheme.

In our experiments, both schemes gave similar results. For this reason, we will only present results for the first scheme.

- *Time of Last Update (time)*: This feature is derived from the time-stamp of each file on the disk. We include only the month and year.

In order to investigate the results LIMBO produced with the aforementioned non-structural features, we consider all possible combinations of them added to the structural information. These combinations are depicted in the lattice of Figure 4. At the bottom of this lattice we have the structural dependencies, and as we follow a path upwards, different non-structural features are added. Thus, in the first level of the lattice, we only add individual non-structural features. Each addition is represented by a different type of arrow at each level of the lattice. For example, the addition of *dir* is given by a solid arrow. As the lattice of Figure 4 suggests, there are fifteen possible combinations of non-structural features that can be added to the structural information.

Each combination of non-structural features in Figure 4 is annotated with the MoJo distance between the decomposition created by LIMBO and the authoritative one. The results are also given, in ascending order of the MoJo distance value, in Table 9. The table also includes the number of clusters that the proposed decomposition had in each case.

	Clusters	MoJo
dev+dir	69	178
dev+dir+time	37	189
dir	25	195
dir+loc+time	78	201
dir+time	18	208
dir+loc	74	210
dev+dir+loc	49	212
dev	71	229
structural	56	237
time	66	239
dev+time	73	240
dev+loc	73	242
dev+loc+time	45	248
dev+dir+loc+time	48	248
loc+time	34	265
loc	85	282

Table 9. Number of clusters and MoJo distance between the proposed and the authoritative decomposition.

The first observation to be made is that certain combinations of non-structural data produce clusterings with a smaller MoJo distance to the authoritative decomposition than the clustering produced when using structural input. This indicates that the inclusion of non-structural data has the potential to increase the quality of the obtained decomposition. However, in some of the cases the MoJo distance to the authoritative decomposition has increased significantly.

A closer look reveals some interesting trends:

- Following a solid arrow in the lattice always leads to a smaller MoJo value (with the exception of the topmost one where the value is actually the same). This indicates that the inclusion of directory structure information produces better decompositions, an intuitive result.

- Following a dashed arrow leads to a smaller MoJo value as well, although the difference is not as dramatic as before (the topmost dashed arrow is again an exception). Still, this indicates that ownership information has a positive effect on the obtained clustering, a result that confirms the findings of Holt and Bowman [6].

- Following a dotted arrow consistently decreases the quality of the obtained decomposition (a marginal exception exists between *dir+time* and *dir+loc+time*). This confirms our expectation that using the lines of code as a basis for software clustering is not a good idea.

- Finally, following the arrows that indicate addition of *time*, leads mostly to worse clusterings but only marginally. This indicates that time could have merit as a clustering factor, but maybe in a different setting. It is quite possible that if we obtain information about which files are being developed around the same time by examining the revision control logs of a system, that we will get better results.

A further observation is that the few exceptions to the above trends that we encountered occurred in the top part of the lattice. This can probably be attributed to the fact that when a number of factors has already been added to the algorithm's input, the effect of a new factor will not be as significant, and in fact it might be eclipsed by the effect of other factors. As a result, we believe that the lower part of the lattice yields more accurate results than the top part.

It is interesting to note that when the structural information was removed from LIMBO's input, the results were not as good. In fact, *loc* and *time* produced rather random decompositions. The situation was better for *dir* and *dev* (MoJo distances to the authoritative decomposition of 407 and 317 respectively), but still quite far from the results obtained from the combination of structural and non-structural data. This result indicates that an effective clustering algorithm needs to consider both structural and non-structural information in order to produce decompositions that are close to the conceptual architecture of a software system.

Finally, the execution times observed for these experiments were almost identical to the ones reported in Section 4.

In summary, the results of our experiments show that directory structure and ownership information are important factors for the software clustering process, while lines of code is not. Temporal information might require more careful setup before

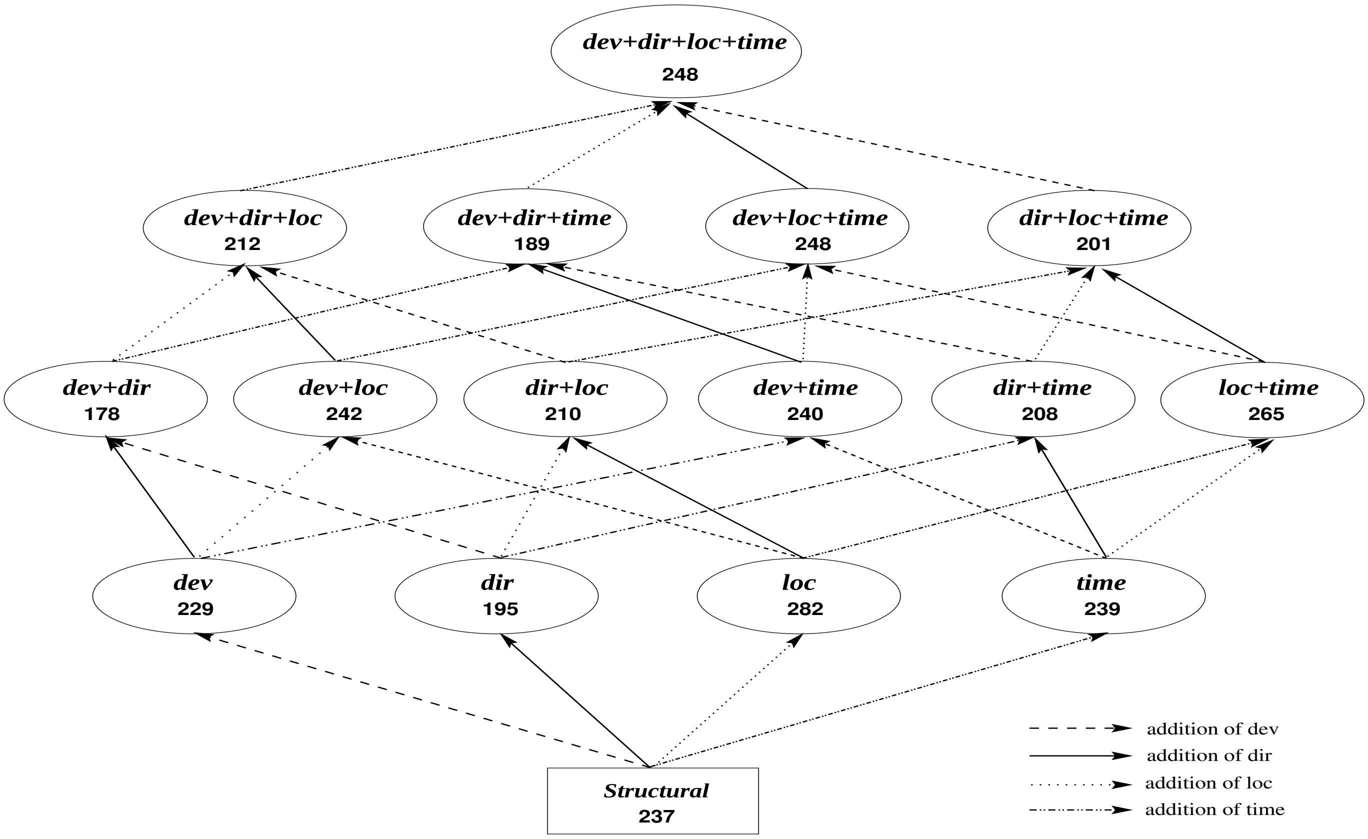

Figure 4. Lattice of combinations of non-structural features for the Linux system

it can be effective. Further research is, of course, required in order to determine whether these results hold true for a variety of software systems, or are particular to the Linux kernel.

6 Conclusions

This paper presented the novel notion that information loss minimization is a valid basis for a software clustering approach. We developed an algorithm that follows this approach and showed that it performs as well, if not better than existing algorithms.

Our approach has the added benefit that it can incorporate in the software clustering process any type of information relevant to the software system. We experimented and assessed the usefulness of four different such types of information.

Certain avenues for further research present themselves. Further experimentation with more software systems is one of our plans. The decompositions created by LIMBO will also have to be evaluated in an empirical study, where developers of the clustered systems provide feedback on them. We are definitely excited to investigate other types of information that are potentially useful to the software clustering process. Such types include date of creation, revision control logs, as well as concepts extracted from the source code using various concept analysis techniques. Finally, applying relative weights to the various types of data used as input, is also a possibility for future work.

Acknowledgements

We would like to thank Nicolas Anquetil for providing the implementation of the cluster analysis algorithms, as well as Spiros Mancoridis for the Bunch tool. We are also grateful to Ivan Bowman and R.C. Holt for the developer information of the Linux kernel. Finally, we thank Panayiotis Tsaparas, Renée J. Miller and Kenneth C. Sevcik for their contribution to the development of LIMBO.

This work was supported in part by the National Sciences and Engineering Research Council of Canada (NSERC).

References

[1] P. Andritsos and R. J. Miller. Reverse Engineering meets Data Analysis. In *Proceedings of the Ninth International Workshop on Program Comprehension*, pages 157–166, May 2001.

[2] P. Andritsos, P. Tsaparas, R. J. Miller, and K. C. Sevcik. Limbo: A scalable algorithm to cluster categorical data.

Technical report, UofT, Dept of CS, CSRG-467, 2003.

[3] N. Anquetil and T. Lethbridge. File clustering using naming conventions for legacy systems. In *Proceedings of CASCON 1997*, pages 184–195, Nov. 1997.

[4] N. Anquetil and T. Lethbridge. Experiments with clustering as a software remodularization method. In *Proceedings of the Sixth Working Conference on Reverse Engineering*, pages 235–255, Oct. 1999.

[5] I. T. Bowman and R. C. Holt. Software architecture recovery using conway's law. In *Proceedings of CASCON 1998*, pages 123–133, Nov. 1998.

[6] I. T. Bowman and R. C. Holt. Reconstructing ownership architectures to help understand software systems. In *Proceedings of the Seventh International Workshop on Program Comprehension*, May 1999.

[7] I. T. Bowman, R. C. Holt, and N. V. Brewster. Linux as a case study: Its extracted software architecture. In *Proceedings of the 21th International Conference on Software Engineering*, May 1999.

[8] S. C. Choi and W. Scacchi. Extracting and restructuring the design of large systems. *IEEE Software*, pages 66–71, Jan. 1990.

[9] T. M. Cover and J. A. Thomas. *Elements of Information Theory*. Wiley & Sons, New York, NY, USA, 1991.

[10] M. R. Garey and D. S. Johnson. *Computers and intractability; a guide to the theory of NP-completeness*. W.H. Freeman, 1979.

[11] D. H. Hutchens and V. R. Basili. System structure analysis: Clustering with data bindings. *IEEE Transactions on Software Engineering*, 11(8):749–757, Aug. 1985.

[12] R. Koschke. Atomic architectural component recovery for program understanding and evolution. *Ph.D. Thesis, Institute for Computer Science, University of Stuttgart*, 2000.

[13] R. Koschke and T. Eisenbarth. A framework for experimental evaluation of clustering techniques. In *Proceedings of the Eighth International Workshop on Program Comprehension*, pages 201–210, June 2000.

[14] R. Lutz. Recovering high-level structure of software systems using a minimum description length principle. In *Proceedings of the 13th Irish Conference on Artificial Intelligence and Cognitive Science*, Sept. 2002.

[15] S. Mancoridis, B. Mitchell, Y. Chen, and E. Gansner. Bunch: A clustering tool for the recovery and maintenance of software system structures. In *Proceedings of the International Conference on Software Maintenance*. IEEE Computer Society Press, 1999.

[16] E. Merlo, I. McAdam, and R. D. Mori. Source code informal information analysis using connectionist models. In *International Joint Conference on Artificial Intelligence*, pages 1339–1344, 1993.

[17] B. S. Mitchell and S. Mancoridis. Comparing the decompositions produced by software clustering algorithms using similarity measurements. In *Proceedings of the International Conference on Software Maintenance*, pages 744–753, Nov. 2001.

[18] H. A. Müller, M. A. Orgun, S. R. Tilley, and J. S. Uhl. A reverse engineering approach to subsystem structure identification. *Journal of Software Maintenance: Research and Practice*, 5:181–204, Dec. 1993.

[19] N. Slonim and R. Somerville and N. Tishby and O. Lahav. Objective Classification of Galaxies Spectra using the Information Bottleneck Method. *Monthly Notices of the Royal Astronomical Society, (MNRAS)*, 323(270), 2001.

[20] R. W. Schwanke. An intelligent tool for re-engineering software modularity. In *Proceedings of the 13th International Conference on Software Engineering*, pages 83–92, May 1991.

[21] R. W. Schwanke and M. A. Platoff. Cross references are features. In *Second International Workshop on Software Configuration Management*, pages 86–95. ACM Press, 1989.

[22] N. Slonim and N. Tishby. Agglomerative Information Bottleneck. In *Neural Information Processing Systems, (NIPS-12)*, pages 617–623, 1999.

[23] N. Slonim and N. Tishby. Document Clustering Using Word Clusters via the Information Bottleneck Method. In *Proceedings of the 23rd Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 208–215, 2000.

[24] N. Tishby, F. C. Pereira, and W. Bialek. The Information Bottleneck Method. In *37th Annual Allerton Conference on Communication, Control and Computing*, Urban-Champaign, IL, 1999.

[25] V. Tzerpos and R. C. Holt. MoJo: A distance metric for software clusterings. In *Proceedings of the Sixth Working Conference on Reverse Engineering*, pages 187–193, Oct. 1999.

[26] V. Tzerpos and R. C. Holt. ACDC: An algorithm for comprehension-driven clustering. In *Proceedings of the Seventh Working Conference on Reverse Engineering*, pages 258–267, Nov. 2000.

[27] Z. Wen and V. Tzerpos. An optimal algorithm for MoJo distance. In *Proceedings of the Eleventh International Workshop on Program Comprehension*, pages 227–235, May 2003.

[28] T. Zhang, R. Ramakrishnan, and M. Livny. BIRCH: An efficient Data Clustering Method for Very Large Databases. In *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, pages 103–114, June 1996.