

Comparison of Data Structures for Computing Formal Concepts^{*}

Petr Krajca^{1,2} and Vilem Vychodil^{1,2}

¹ T. J. Watson School, State University of New York at Binghamton

² Dept. Computer Science, Palacky University, Olomouc
{petr.krajca, vychodil}@binghamton.edu

Abstract. Presented is preliminary study of the role of data structures in algorithms for formal concept analysis. Studied is performance of selected algorithms in dependence on chosen data structures and size and density of input object-attribute data. The observations made in the paper can be seen as guidelines on how to select data structures for implementing algorithms for formal concept analysis.

Keywords: formal concept analysis, data structures, algorithms, performance, comparison.

1 Introduction

Formal concept analysis (FCA) [17] is a method of qualitative data analysis with a broad outreach to other analytical disciplines. Formal concepts, i.e. maximal rectangular submatrices of Boolean object-attribute matrices, which are the basic patterns studied by formal concept analysis, are important for various data-mining and decision-making tasks. For instance, formal concepts can be used to obtain nonredundant association rules [18] and minimal factorizations of Boolean matrices [2]. Recently, it has been shown in [1] that formal concepts can be used to construct decision trees. From the computational point of view, these applications of FCA depend on algorithms for computing all formal concepts (possibly satisfying additional constraints) given an input data set. It is therefore important to pay attention to algorithms for FCA especially in case of large input data where the performance of algorithms becomes a crucial issue.

In this paper we focus on the data structures used in algorithms for computing formal concepts. Selection of the appropriate data structure has an important impact virtually on any algorithm and the decision of which structure is the optimal one for given algorithm is often uneasy. Usually, the decision depends on many factors, especially on the data being processed. Such decision has to be done wisely since selection of an inappropriate structure may lead to a poor performance or to an excessive use of resources in real programs. Algorithms for computing formal concepts are not an exception. Moreover, the situation is complicated since algorithms are usually described in pseudocode that is a

^{*} Supported by institutional support, research plan MSM 6198959214.

language combining (vague) human and (formal) programming languages. This gives certain freedom to a programmer but with this freedom is tightly coupled a big piece of responsibility. For example, if a description of an algorithm contains the term “store $B \cap C$ into A ”, from the point of view of the algorithm description, the statement is clear and sufficiently descriptive. The term says: “store intersection of sets B and C into set A ”. On the other hand, from the implementation point of view, such description is ambiguous because it does not provide any information on how such intersection should be computed and how the sets should be represented.

Interestingly, the data representation issues are almost neglected in literature on FCA. The well-known comparison study [11] of FCA algorithms mentions the need to study the influence of data structures on practical performance of FCA algorithms but it does not pay attention to that particular issue. This paper should be considered a first step towards this direction. Recall that the limiting factor of computing all formal concepts is that the problem is $\#P$ -complete [8]. The theoretical complexity of algorithms for FCA is usually expressed in terms of time delay [7] and all commonly used FCA algorithms have polynomial time delay [8]. Still, the asymptotic complexity does not say which algorithm is faster as many different algorithms belong to the same class. With various data structures, the problem becomes even more complicated. Therefore, there is a need for experimental evaluation which may help users decide which FCA algorithm should be used for particular type of data, cf. [11]. In this paper, we try to answer a related question: “Which data representation should be chosen for particular type of data?”

The paper is organized as follows. Section 2 presents a survey of notions of FCA and used algorithms. Section 3 describes used data structures and set-theoretical operations on these data structures. Finally, Section 4 presents experimental evaluation showing the impact of data structures on the performance and concluding remarks.

2 Formal Concept Analysis

In this section we recall basic notions of the formal concept analysis (FCA). More details can be found in monographs [6] and [3].

2.1 Survey of Basic Notions

FCA deals with binary data tables describing relationship between objects and attributes, respectively. The input for FCA is a data table with rows corresponding to objects, columns corresponding to attributes (or features), and table entries being 1’s and 0’s, indicating whether an object given by row has or does not have an attribute given by column. The input is formalized by a binary relation $I \subseteq X \times Y$, $\langle x, y \rangle \in I$ meaning that object x has attribute y , and I being called a *formal context* [6]. Each formal context $I \subseteq X \times Y$ induces a couple of concept-forming operators \uparrow and \downarrow defined, for each $A \subseteq X$ and $B \subseteq Y$, by

$$A^\uparrow = \{y \in Y \mid \text{for each } x \in A: \langle x, y \rangle \in I\}, \quad (1)$$

$$B^\downarrow = \{x \in X \mid \text{for each } y \in B: \langle x, y \rangle \in I\}. \quad (2)$$

Operators $\uparrow: 2^X \rightarrow 2^Y$ and $\downarrow: 2^Y \rightarrow 2^X$ defined by (1) and (2) form a so-called Galois connection [6]. By definition (1), A^\uparrow is a set of all attributes shared by all objects from A and, by (2), B^\downarrow is a set of all objects sharing all attributes from B . A pair $\langle A, B \rangle$ where $A \subseteq X$, $B \subseteq Y$, $A^\uparrow = B$, and $B^\downarrow = A$, is called a *formal concept* (in $I \subseteq X \times Y$). Formal concepts can be seen as particular clusters hidden in the data. Namely, if $\langle A, B \rangle$ is a formal concept, A (called an *extent* of $\langle A, B \rangle$) is the set all objects sharing all attributes from B and, conversely, B (called an *intent* of $\langle A, B \rangle$) is the set of all attributes shared by all objects from A . From the technical point of view, formal concepts are fixed points of the Galois connection $\langle \uparrow, \downarrow \rangle$ induced by I . Formal concepts in $I \subseteq X \times Y$ correspond to so-called maximal rectangles in I . In a more detail, any $\langle A, B \rangle \in 2^X \times 2^Y$ such that $A \times B \subseteq I$ shall be called a rectangle in I . Rectangle $\langle A, B \rangle$ in I is a maximal one if, for each rectangle $\langle A', B' \rangle$ in I such that $A \times B \subseteq A' \times B'$, we have $A = A'$ and $B = B'$. We have that $\langle A, B \rangle \in 2^X \times 2^Y$ is a maximal rectangle in I iff $A^\uparrow = B$ and $B^\downarrow = A$, i.e. maximal rectangles = formal concepts.

The set of all formal concepts in I is denoted by $\mathcal{B}(X, Y, I)$. In this paper, we will be interested in performance of algorithms computing (listing all concepts in) $\mathcal{B}(X, Y, I)$. Note that $\mathcal{B}(X, Y, I)$ can optionally be equipped with a partial order \leq modeling the subconcept-superconcept hierarchy: We put $\langle A_1, B_1 \rangle \leq \langle A_2, B_2 \rangle$ iff $A_1 \subseteq A_2$ (or, equivalently, iff $B_2 \subseteq B_1$). If $\langle A_1, B_1 \rangle \leq \langle A_2, B_2 \rangle$ then $\langle A_1, B_1 \rangle$ is called a subconcept of $\langle A_2, B_2 \rangle$. The set $\mathcal{B}(X, Y, I)$ together with \leq form a complete lattice whose structure is described by the Main Theorem of Formal Concept Analysis [6].

2.2 Algorithms for Computing Formal Concepts

Several algorithms for computing formal concepts have been proposed. In our experiments, we have considered three well-known algorithms—Ganter’s NextClosure [5], Lindig’s UpperNeighbor [13], and Kuznetsov’s CloseByOne [9,10] which is conceptually close to the algorithm of Norris [14]. These algorithms are commonly used for computing formal concepts and, therefore, their efficient implementation is crucial.

A detailed description of the algorithms is outside the scope of this paper. Interested readers can find details in the papers cited above and in a survey paper [11] presenting a comparison of various algorithms for FCA. Just to recall, NextClosure and CbO are algorithms which are conceptually close because they perform the same canonicity test to prevent listing the same formal concept multiple times. The fundamental difference of the algorithm is in the strategy in which they traverse through the search space containing all formal concepts. Although mutually reducible, the algorithms are different from the practical efficiency point of view as we will see later and as it is also shown in [11]. Lindig’s algorithm belongs to a different family of algorithms that compute

formal concepts and the subconcept-superconcept ordering \leq at the same time. The algorithm keeps track of all formal concepts that have been computed, i.e. it stores them in a data structure. Usually, a balanced tree or a hash table is used to store concepts. The concepts are stored in a data structure for the sake of checking whether a formal concept has been found in previous steps of computation.

2.3 Representation of Formal Contexts and Computing Closures

Representation of the input data (a formal context) is crucial and has an important impact on performance of real applications. To increase the speed of our implementations, we store each context in two set-theoretical forms. This allows us to (i) increase speed of computing closures for certain algorithms and (ii) we are able to use a uniform data representation for contexts, extents, and intents. The first form is an array of sets containing, for each object $x \in X$, a set $\{x\}^\uparrow$ of all attributes of object x . Dually, the second form is an array of sets containing, for each attribute $y \in Y$, a set $\{y\}^\downarrow$ of all objects having attribute y . This redundant representation of contexts can significantly improve the speed of UpperNeighbor and CbO. Namely, given a formal concept $\langle A, B \rangle$ and $y \notin B$, we can compute a new formal concept $\langle A \cap \{y\}^\downarrow, (A \cap \{y\}^\downarrow)^\uparrow \rangle$ by intersecting sets of objects and attributes from both context representations [15].

3 Used Data Structures and Algorithms: An Overview

The most critical operations used in the algorithms for computing formal concepts are set operations and predicates that are needed to manipulate extents and intents of computed formal concepts. This means, operations of *intersection*, *union*, *difference* and predicate of *membership* (\cap , \cup , \setminus , and \in , respectively). Therefore, we focus on data structures that allow to efficiently implement these operations. In the sequel, we provide a brief overview of five data structures we deem suitable to represent sets and which will be used in our performance measurements.

3.1 Bit Array

The first data structure we use to represent a set is an array of bits. A bit array is a sequence of 0's and 1's and is very suitable for representing characteristic function of a set. If the element is present in a set, the bit at the appropriate position has value 1, otherwise it has value 0. Let us consider universe $U = \{a, b, c, d, e\}$ and bit array 01001 such bit array may represent set $\{b, e\}$. Obviously, one has to fix a total order on U in order to make such representation unambiguous. In the sequel, we are going to use a set $X = \{0, 1, \dots, m\}$ of objects and a set $Y = \{0, 1, \dots, n\}$ of attributes, respectively, with the natural ordering of numbers. In other words, each element in X or Y can be used as an index (in a bit array). Note that there is no danger of confusing objects with attributes because we do not mix elements from the sets X and Y in any way.

An important feature of this data structure is that all operations may be reduced to few bitwise operations performed directly by CPU. For instance, if we consider two sets from universe of size 64, their intersection may be computed on contemporary computers in one operation (bitwise logical AND). On the other hand, the size of the data structure representing a set is determined by the size of the universe and not by the size of the set itself. This may be a serious disadvantage while dealing with small sets defined in large universes—a situation that may frequently occur when dealing with sparse data sets with low densities of 1's (i.e., low percentages of 1's in the context, meaning that $|I|$ is small compared to $|X| \cdot |Y|$). In such a case, sets occupy large segments of memory and operations may not have to be so efficient as expected.

3.2 Sorted Linked List

Linked lists represent another type of a data structure suitable and frequently used for representing sets. The usage is obvious, an element belongs into a set, if and only if it is present in the list representing the set (we allow no element duplicities in lists). In the sequel, we consider a variant of linked list, where all elements are sorted w.r.t. the fixed total order (see Section 3.1). This allows us to implement set operations more efficiently. For instance, while computing an intersection of two sets, we may use so called *merging*. This means, we take both lists and repeatedly apply the following procedure:

If the first elements of both the lists are the same, we put this element into the resulting set and we remove both the elements from the considered lists, otherwise we remove the least element.

We repeat this procedure until one of the lists is empty and then the resulting set contains only the elements that are present in both sets. Other set operations can be implemented analogously taking into account the total ordering of elements in the lists.

From the point of view of memory requirements, linked lists have certain overhead since with each element of a list we have to allocate an additional space for pointer to the next element of the list.

3.3 Array

In much the same way as in case of lists, we may store elements of a set into an array. This makes the representation of a set more space efficient since we do not have to store pointers between elements. Furthermore, if the set elements are ordered, we may optimize particular operations with the *divide et impera* approach by employing binary search [12]. For instance, to compute intersection of two sets we can go through all elements in the smaller set and using binary search we can check whether given element is also in the second set.

On the other hand, the advantages of arrays are counterweighted by the fact that arrays need some additional effort to shrink or expand their size while adding or removing elements from the set.

3.4 Binary Search Tree

Binary search tree is a data structure that has similar time complexity of the essential operations as an array. For example, when computing intersection of two sets, we proceed in a similar way as in case of arrays. We go through all elements in the smaller set and check if the elements is also in the second one. If the tree is balanced, we can do such check in a logarithmic time. This means, computation of the intersection has time complexity $O(n \cdot \log m)$.

Besides the performance, other advantages of binary trees include more efficient insertion and deletion of elements than in case of arrays. On the other hand, trees are less space efficient and need additional effort to keep them balanced to provide adequate performance. Several variants of binary search trees were proposed. For our experiments we have selected self-balancing lean-left red-black tree [4,16] for its efficiency and brevity of its implementation.

3.5 Hash Table

The last data structure we consider is a hash table. The hash tables are usually not so space efficient as the previously mentioned data structures but the time complexity of operations with hash tables is comparable. For example, computation of intersection is similar as in the previous cases: We go through all elements in the first set and check whether the elements are also in the second one. We have included hash tables since they are frequently used to implement sets in standard libraries of programming languages. There is therefore a temptation of using such library structures for representing sets. In our experiments, we have considered a variant of hash table with separate chaining [4].

3.6 Time Complexity of Operations

Fig. 1 depicts asymptotic time complexities of the elementary set operations with respect to data structures representing sets. The the time complexities are expressed in terms of the O -notation.

	\cap	\cup	\in
bit array	$O(U)$	$O(U)$	$O(1)$
sorted linked list	$O(m + n)$	$O(m + n)$	$O(n)$
sorted array	$O(n \cdot \log m)$	$O(n \cdot \log m)$	$O(\log n)$
binary search tree	$O(n \cdot \log m)$	$O(n \cdot \log m)$	$O(\log n)$
hash table	$O(m \cdot n)$	$O(m \cdot n)$	$O(n)$

Fig. 1. Worst-case time complexity of set operations

Remarks. The O -notation captures only a particular aspect of the time complexity and real performance of data structures may significantly differ. Sometimes, it may be useful to join several elementary operations into a single compound operation. For instance, Ganter’s algorithm and CbO perform a canonicity test that prevents computing concepts multiple times. This test consists of several operations

and it seems to be practical to implement this test as one operation that takes advantage of the underlying data structure. For example, in the CloseByOne (CbO) algorithm, the test is defined as $B \cap Y_j = D \cap Y_j$, where $D = (B \cup \{y\})^{\downarrow\uparrow}$ is an intent of a newly generated concept, B is the intent of previously generated concept, and Y_j represents first j attributes. In some cases, it may be inefficient to compute the intersections of sets B and D with the set Y_j and then compare the results. In fact, it suffices to compare just one set inclusion $B \cap Y_j \supseteq D \cap Y_j$ as the converse inclusion follows from the monotony of the closure operator $\downarrow\uparrow$. In other cases, however, it may be more efficient to perform the test without computing the intersections first: we check whether each $y \in D$ such that $y \leq j$ is present in B . From the point of view of the asymptotic complexity, such optimization is not essential because its complexity is the same. On the other hand, the impact on the practical performance of such optimization may be significant as we will see in the next section.

4 Experimental Performance Measurements

In this section we discuss the behavior of algorithms under various conditions. This means, we have tested algorithms for various input datasets (with various sizes and densities) and data structures.

4.1 Implementation

In order to compare properties of algorithms and data structures, we have implemented all tests in the C language. All programs share the same code base, i.e., the implementation of each algorithm is shared and only the implementation of (operations with) data structures differs. We have almost directly translated the usual pseudocode of algorithms to an equivalent code in C. In the codes of the algorithms, we have not employed any specific optimizations but to emulate environment of real applications and to reflect strengths of each data structure, we have optimized particular operations. For instance, while computing the outcomes of the concept-forming operators \downarrow and \uparrow , it is necessary to compute an intersection of multiple sets, see Section 2.3. One option to compute such intersection is to repeatedly apply an operation of intersection of two sets. This approach is for example suitable for bit arrays. On the other hand, in some cases it is possible to compute intersection of multiple sets more efficiently if we consider all sets in the intersection. For instance, this applies for ordered lists, i.e., we perform a merge of all the lists simultaneously.

Remarks. The C language has been selected for testing purposes since it allows equally efficient implementations of all considered data structures. If anyone is going to use other programming language, he or she should be aware of its specific properties. For instance, in case of Java or C#, particular data structures may be less efficient due to issues connected to auto-boxing, etc.

All experiments were done on an otherwise idle computer equipped with two quad-core Intel Xeons E5345, 12 GB RAM, GNU/Linux and we have compiled all programs with GCC 4.1.2 with only `-O3` option turned on.

4.2 Performance

In our experiments, we compare running times needed to compute all formal concepts using considered algorithms with particular data structures. Since the time of computation is dependent on many factors, especially the size of the data

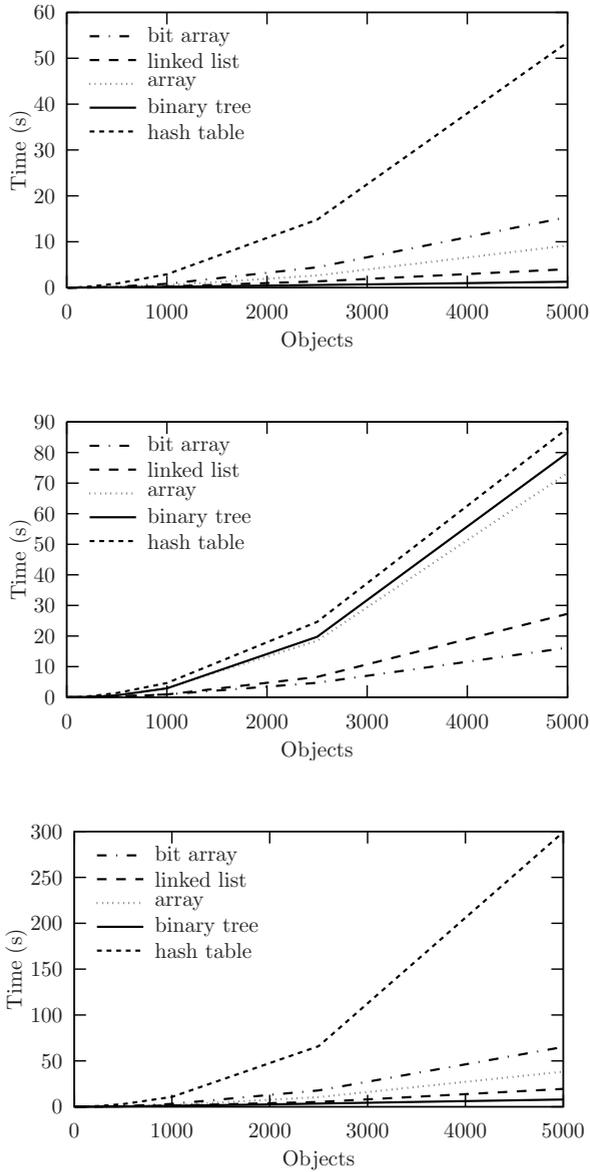


Fig. 2. Efficiency of data structures for particular algorithms—CloseByOne (top); NextClosure (middle); UpperNeighbor (bottom)

and density of 1's present in the data matrix, we have used randomly generated data tables of various properties for our experiments.

In the first set of experiments, we tried to answer the question if some structure is better for particular algorithm then other structures. To find the answer, for each algorithm we compared time it takes to compute all formal concepts in data tables with various numbers of objects, 50 attributes, where the density of 1's in the data table is 15%. We have selected 15% because data used in formal concept analysis are usually sparse (but there can be exceptions). The results are presented in Fig. 2. One can see that for CloseByOne and Lindig's UpperNeighbor, the tree representation of sets is the optimal one and for Ganter's NextClosure it is the bit array representation. Notice that the linked list representation provides reasonable results for all algorithms. On the contrary, hash table representation seems to provide a poor performance under all circumstances.

The previous experiment involved a fixed number of attributes and a fixed density of 1's in the data matrix. Since the dimension and density of contexts have considerable impact on performance, we performed additional experiments where the dimensions and/or density are variable.

In the next experiment, we selected tables of size 100×100 and 500×100 with various densities of 1's and compared time it takes to compute all formal concepts. Fig. 3 and Fig. 4 show times for the CloseByOne algorithm (the results for other algorithms turned out to be analogous and are therefore omitted). Notice, that we have used logarithmic scale for the time axes. This allows us to identify a point where some data structures become less efficient. Fig. 3 and Fig. 4 also indicate that linked list, binary tree and array are suitable for sparse data. Contrary to that, bit arrays are more suitable for dense data.

The point, i.e., the density, for which bit array outperforms other representations is dependent on other factors. As we can see in Fig. 3 and Fig. 4, for larger data tables this point shifts to higher densities.

The last property of data tables we are going to consider is the number of attributes. The results are shown Fig. 5, presenting computation time of NextClosure algorithm with data tables consisting of 100 objects, various numbers of attributes and 15% density of 1's. With the exception of hash tables and binary

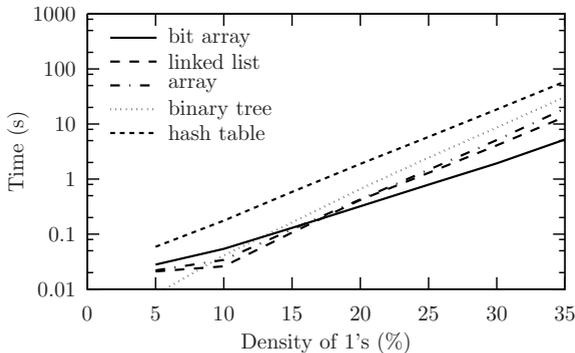


Fig. 3. Efficiency of data structures for various densities; table 100×100

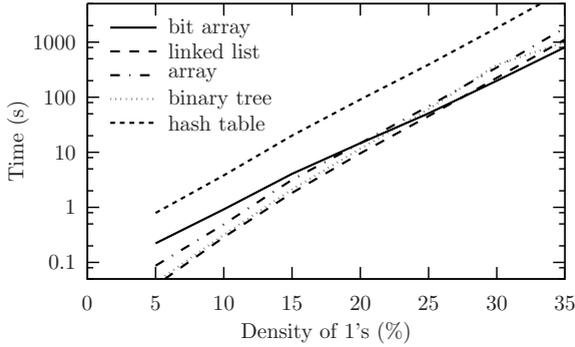


Fig. 4. Efficiency of data structures for various densities; table 500×100

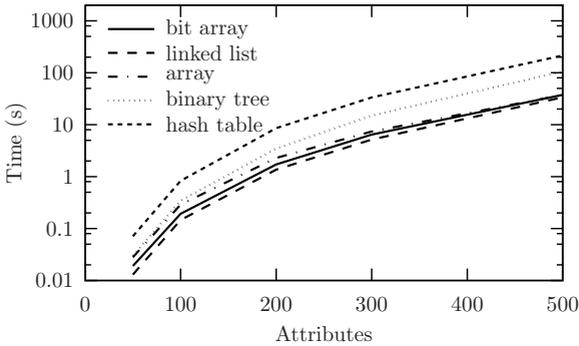


Fig. 5. Efficiency of data structures for various numbers attributes in table with 100 objects

trees, the number of attributes does not have a significant impact on the time of the computation. Notice that in case of bit arrays, linked lists, and arrays, the impact is so insignificant that we had to use logarithmic scale to make the corresponding lines distinct.

4.3 Memory Efficiency

In previous sections, we have focused on time efficiency of algorithms and data structures. Another important aspect of data structures is their space efficiency. Common feature of the Ganter's NextClosure and Kuznetsov's CloseByOne algorithms is that they are not extensively memory demanding. NextClosure has a constant memory complexity and CloseByOne has a (worst-case) linear memory complexity depending on the number of attributes (in practice, the memory consumption is typically strongly sublinear).

This means, the size of particular data structure chosen for representing sets affects practically just the size of the (the representation of) the context. The size of (the representation of) the context does not have significant influence on the

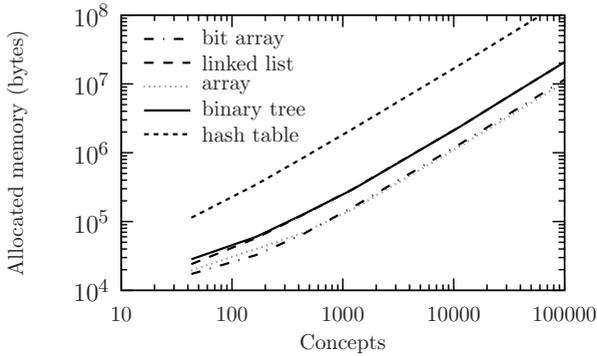


Fig. 6. Memory consumption

overall memory consumption. On the other hand, Lindig’s UpperNeighbor needs to store generated concepts (or at least their intents or extents) to check whether a newly computed concept have already been generated or not. This feature may seriously affect the size of data that may be processed by this algorithm, i.e., all concepts present in the data have to fit into available memory.

In the following experiment, we have focused on the memory consumption. We have selected random data matrix with 100 attributes, various counts of objects and density of 1’s 10%. Fig. 6 shows the growth of the allocated memory dependent on the number of concepts present in the data. One can see that the bit array and array representations require approximately the same amount of memory. Furthermore, this applies also for linked list and binary tree representations. The disproportion between the assumed memory consumption and the real one may be caused by the memory management. Memory allocators in modern operating systems usually do not allocate the exact amount of memory that is requested for an object but allocate rather a larger amount.

Conclusions

This paper addresses an important but overlooked issue: which data structures should be chosen to compute formal concepts. As expected, there is no “the best” structure suitable for all types of data tables and data structures have to be wisely selected. The paper provides a survey with guidelines on how to select such data structure in dependence on the data size, used algorithm, and density. It contains our initial observations on the role of data structures in FCA in terms of the efficiency. If your data is sparse or if you have to deal with large dataset, binary search trees or linked lists are good choices. If you have dense data or smaller data table, the bit array seems to be an appropriate structure. Definitely, usage of hash tables should be avoided as it has shown to be inappropriate for computing formal concepts.

Future research will focus on considering more data structures, mixed data representations, and statistical description of factors and conditions that may have a (hidden) influence on the choice of data structure.

References

1. Belohlavek, R., De Baets, B., Outrata, J., Vychodil, V.: Inducing decision trees via concept lattices. *Int. J. General Systems* 38(4), 455–467 (2009)
2. Belohlavek, R., Vychodil, V.: Discovery of optimal factors in binary data via a novel method of matrix decomposition. *Journal of Computer and System Sciences* (to appear)
3. Carpineto, C., Romano, G.: *Concept data analysis. Theory and applications*. J. Wiley, Chichester (2004)
4. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: *Introduction to Algorithms*. The MIT Press, Cambridge (2001)
5. Ganter, B.: Two basic algorithms in concept analysis. Technical Report FB4-Preprint No. 831. TH Darmstadt (1984)
6. Ganter, B., Wille, R.: *Formal concept analysis. Mathematical foundations*. Springer, Berlin (1999)
7. Johnson, D.S., Yannakakis, M., Papadimitriou, C.H.: On generating all maximal independent sets. *Information Processing Letters* 27(3), 119–123 (1988)
8. Kuznetsov, S.: Interpretation on graphs and complexity characteristics of a search for specific patterns. *Automatic Documentation and Mathematical Linguistics* 24(1), 37–45 (1989)
9. Kuznetsov, S.: A fast algorithm for computing all intersections of objects in a finite semi-lattice (Быстрый алгоритм построения всех пересечений объектов из конечной полурешетки, in Russian). *Automatic Documentation and Mathematical Linguistics* 27(5), 11–21 (1993)
10. Kuznetsov, S.: Learning of Simple Conceptual Graphs from Positive and Negative Examples. In: Żytkow, J.M., Rauch, J. (eds.) *PKDD 1999. LNCS (LNAI)*, vol. 1704, pp. 384–391. Springer, Heidelberg (1999)
11. Kuznetsov, S., Obiedkov, S.: Comparing performance of algorithms for generating concept lattices. *J. Exp. Theor. Artif. Int.* 14, 189–216 (2002)
12. Levitin, A.V.: *Introduction to the Design and Analysis of Algorithms*. Addison Wesley, Reading (2002)
13. Lindig, C.: Fast concept analysis. In: *Working with Conceptual Structures—Contributions to ICCS 2000*, pp. 152–161. Shaker Verlag, Aachen (2000)
14. Norris, E.M.: An Algorithm for Computing the Maximal Rectangles in a Binary Relation. *Revue Roumaine de Mathématiques Pures et Appliquées* 23(2), 243–250 (1978)
15. Outrata, J., Vychodil, V.: Fast algorithm for computing fixpoints of Galois connections induced by object-attribute relational data (submitted)
16. Sedgewick, R.: Left-leaning red-black trees, <http://www.cs.princeton.edu/~rs/talks/LLRB/LLRB.pdf>
17. Wille, R.: Restructuring lattice theory: an approach based on hierarchies of concepts. In: *Ordered Sets*, Dordrecht-Boston, pp. 445–470 (1982)
18. Zaki, M.J.: Mining non-redundant association rules. *Data Mining and Knowledge Discovery* 9, 223–248 (2004)