Parallel Execution of Binary-Based NextClosure Algorithm

Simin Jabbari and Kilian Stoffel

Information Management Institute, University of Neuchâtel,
Neuchâtel 2000, Switzerland
{simin.jabbari,kilian.stoffel}@unine.ch
https://www2.unine.ch/imi

Abstract. Formal concept analysis (FCA) has become a popular method for analyzing data across various domains in which data bases can be analyzed regardless of their contexts. With its properties FCA is of big interest in the context of Big Data. However, the complexity of the basic FCA analysis algorithms often prohibits its use in general production tool chains for data analysis. In this paper we show how to overcome some of these problems. In the first step we show how to implement the well known NextClosure in efficient way in Python (a preferred language in the context of ad-hoc data analysis) which is several times faster the other published algorithms. In the second step we show how our implementation can be parallelized on common hardware by strictly using the best sequential algorithm which differs in an important way form so far published parallel algorithms for FCA.

Keywords: Formal concept analysis, NextClosure, Parallel programming, Binary computation.

1 Introduction

Nowadays, Big Data has entered to a new era and new data is produced every day. The processing of such huge amounts of data as fast as possible has now become a must. Without having the ability to quickly process newly acquired data, collected by variety of devices and techniques, we can not expect to reach the milestones already set in Big Data.

Formal concept analysis (FCA) [1–3] has become a popular method for analyzing data across various domains including knowledge discovery, data mining, and social networks [4–7]. The significance of FCA for data analysis is that it provides a unique framework in which datasets can be analyzed regardless of their contexts. Using FCA, we can describe the relationship between a set of objects and a set of attributes. One of the key outputs of FCA is the set of formal concepts that mathematically represent the usual notion of concepts. How to extract formal concepts from a dataset has been extensively studied by many researchers [1, 8, 9]. Among the proposed algorithms, such as CbO [10–12], In-Close [13], and UpperNeighbor [14], it has been shown that NextClosure (NC) [1]

is one of the most promising methods as it needs much less time for implementation compared to its counterparts [15]. There are many other implementations that have their advantages, but NextClosure is one of the best general purpose algorithms.

The major idea behind NC is that is does not calculate all the possible intents that could be considered. Based on a clever ordering and based on previously constructed intents the next potentially useful intents are created. In this way a lot of unnecessary works can be avoided. But for the purposes of what we intent to do in this paper, namely speeding up this algorithm using an optimal sequential implementations and parallelization, this very sequential nature is a major challenge.

Nowadays, datasets are much bigger than in the past. Processing huge amounts of data requires fast algorithms, methods and tools that enhance the performance of the existing algorithms. There are multiple dimensions that have to be considered. Very often data analysis on large datasets requires ad-hoc solution. Therefore, key parts of algorithms should be easily implementable or at least easily adjustable. To achieve this goal, flexible programming environments (or prototyping environments) are used. These environments evolved over time, currently the Python programming and scripting environment is indeed one of the most popular ones. However, the flexibility and prototyping characteristics of these environments come at certain price. Most of the straightforward implementations of algorithms are not very efficient during execution.

In this paper we present two aspects of how to overcome the efficiency problems mentioned before. The first one is to implement the algorithms in such a way that they represent as close as possible that mathematical definition of the algorithm and allowing for an easy maintenance. The second aspect is to show how these algorithms can be sped using parallelism by activating all cores of modern CPUs.

We first briefly review the standard NC algorithm which is proposed to be implemented in a set-based manner. We then provide a binary-based version of the implementation that speeds up that algorithm based on set-notations. However, the logic underlying the algorithms did not change at all. We then adjust the binary-based code such that it maximally uses all available cores in multiprocessor machines, without changing the logic of the algorithm. This means that this is not necessarily the best parallel implementation but it maintains the flexibility we requested for the algorithms to maintain the possibility of their easy adjustment to new situations. We show that the parallel computation can significantly speed up the execution even further, specially for large datasets. Finally, we study the effect of the number of processors involved in processing on the time spent for the execution. We show that the time required for the execution reduces as the number of cores increases. This is particularly important for the analysis of Big Data.

Although this paper does not propose a novel algorithm for calculating the formal concepts (it is actually the opposite, we try to stay as close as possible with the formal specification of the NC algorithm), it provides practical messages

about how to efficiently implement NC, and possibly other existing algorithms, with the goal of maximally using all available resources including binary calculation as well as parallel programming.

2 The Theory

There are many theoretical introductions to Formal Concept Analysis. We are following here the notation and presentation given in [3]. The relationship between a set of *objects* X (rows) and a set of *attributes* Y (columns), represented by a binary table I, forms a *formal context* denoted by a triplet $\langle X, Y, I \rangle$. Here, I can be viewed as a $|X| \times |Y|$ binary matrix such that if $I(x, y) = 1$ it means that object $x \in X$ has attribute $y \in Y$. In this paper we express X with $\{1, 2, ..., m\}$ and Y with $\{1, 2, ..., n\}$, indicating that there are m objects and n attributes.

The *up* and *down* operators, respectively denoted by u and d, are two key operators by which we can extract the set of formal concepts embedded in a formal context. The up operator u is applied to a subset $A \subset X$ of objects and returns a subset $A^u \subset Y$ of attributes such that each object $x \in A$ has *all* attributes y within A^u. Similarly, the down operator d is applied to a subset $B \subset Y$ of attributes and provides all the objects $B^d \subset X$ that all attributes within B have in common. A formal concept is a pair of (A, B) such that $A^u = B$ and $B^d = A$. If $\{(A_1, B_1), (A_2, B_2), ..., (A_p, B_p)\}$ denotes the set of all formal concepts, then $\{A_1, A_2, ..., A_p\}$ is called the set of *extents* and $\{B_1, B_2, ..., B_p\}$ is called the set of *intents*.

The idea behind NextClosure (algorithm 1) is that it extracts the set of all intents in a mathematical ordering fashion, known as *lexicographic ordering*. So it does not require to check whether each of 2^n subset of attributes is an intent. Therefore, it takes much less time than the naive approach for extracting all the intents. Note that the number of subsets $B \subset Y$ exponentially increases with the number n of attributes. Moreover, having intents, ordered in lexicographic fashion, facilitates concept *lattice* generation which is not in the scope of this paper (see [3] for more information on FCA and NC).

A naive approach for implementing NC, as presented in algorithm 1, is based on using set operators such as union and intersection. Applying these operators on the sets of integers (here, objects and attributes), could be time-consuming. The implementation of this approach using native Python operators is given in algorithm 2. This algorithm illustrates how next intent B_q is created from B_{q-1}. It is an almost textual translation of the mathematical definition. This implementation is very compact and it is particularly well suited for doing theoretical work. However it is not very efficient from an execution time point of view, as we will show later.

2.1 Binary-based implementation of NextClosure

In the binary-based implementation of NC, we use another property of Python namely integers of arbitrary length. We can easily see that the set-operators can

Algorithm 1 NextClosure [1, 3]

Input: $\langle X, Y, I \rangle$ # formal context
Output: `Int` # set of all intents (initialized as empty set)

1: $B = (\emptyset^d)^u \leftarrow$ # the least intent
2: `Int.Append(`B`)` # store B in the set of intents
3: `while not(`$B = Y$`) do`
4: $B \leftarrow B^+$ # replace the current intent B with the next intent B^+
5: `Int.Append(`B`)` # store the intent
6: `endwhile`

$B^+ = B \oplus i$ where i is the greatest one for which $B <_i B \oplus i$
$B \oplus i = \left((B \cap \{1, 2, ..., i-1\} \cup \{i\})^d\right)^u$
$B <_i B' \iff i \in B' \setminus B$ and $B \cap \{1, 2, ..., i-1\} = B' \cap \{1, 2, ..., i-1\}$

Algorithm 2 Set-based computation of next intent from previous intent

```
1: def NextIntent(B): # returns the next intent B+ from the previous intent B
2:   for i from n-1 to 0:
3:     res = set(range(i)).intersection(B) # computes B ∩ {0, 1, ..., i − 1}
4:     B’ = Up(Down(res.add(i))) # returns B ⊕ i (full detail is skipped)
5:     if LexicoLessThan(B,B’,i): return B’ # return B′ if B <_i B′
-------------------------------------------------------------------------
6: def LexicoLessThan(B,B’,i): # check whether B <_i B′
7:   if i not in B’-B: return False # to see if i ∈ B′ \ B
8:   h = set(range(i))
9:   if h.intersection(B) == h.intersection(B’): return True
10:  else: return False
```

be implemented as logical operators on bit-string which are representing set-membership relations. In this implementation an object $x \in X$ having attributes $y \in Y$ is represented by a bit-string where each 1 represents the presents of an attribute and 0 the non-presence. Therefore, an object $x \in X$ can be represented as a long integer in Python. This way we can speed up the execution of NC by skipping unnecessary computations that are imposed by the nature of set-based calculations.

The first step for binary-based computation of NC is to interpret each binary row/column of the table I as an integer number expressed in the base-2 numeral system. For instance, if the first object (corresponding to the first row of table I) is identified by $[1, 0, 1, 1, 0, 0]$, we view it as number $2^0 + 2^2 + 2^3 = 13$, because the attributes of this object include $0, 2, 3$. Note that in contrary to the standard notion, in which the attributes are indexed from 1 to n, in the binary version we index the attributes from 0 to $n - 1$ as it makes computation easier.

A formal context $\langle X, Y, I\rangle$ in the binary-based approach is transformed into two vectors *rows* and *cols* with m and n elements, respectively. Each element of *rows* corresponds to one row of the table I indicated by a number. For example, the number $2^n - 1$ corresponds to an object which has all the attributes and number 0 corresponds to an object which does not have any attributes. Similarly, each element of *cols*, corresponding to a column of table I, lies in the range 0 and $2^m - 1$.

Such a representation makes many computations easier. For instance, the intersection of the current intent B with $\{0, ..., i - 1\}$ can be easily implemented in binary representation by calculating the remainder of the division of B by 2^i. Union operator $\cup$ with an attribute $\{i\}$ which is currently not present can also be easily implemented by adding 2^i. The algorithm 3 summarizes functions that we used for NC implementation in the binary approach.

2.2 Parallelization of the binary-based code

To show the implementation of a parallel version of the binary-based version of our NC algorithm we decided to illustrate how to parallelize the *up* operator. This parallelization is straight forward as the Python code shows.

Algorithm 3 shows that the *up* operator requires an iteration over all the objects corresponding to the rows of the binary table for a given formal context. For large datasets, this iteration could be very time-consuming. Using all available cores can speed up this operation (that is frequently used in NC implementation) by simply asking each core to iterate over a subset of rows. In other words, we split the binary representation in equal chunks and sending them to workers. Once the processing of each worker is finished, the results of all workers are merged to return the final output. This simple technique can significantly speed up the execution time. The python-code for how the up operator can be parallelized in the binary-based version is given in algorithm 4. Although the full details of implementation is not provided, it gives the idea of how to parallelize the up operator in the NC algorithm.

Algorithm 3 Functions used in NC - Binary approach - Python notation

1: `def Up(A):` # up operator, A is a number representing a subset of objects.
2: `r = 2**n - 1` # assume that objects within A have all attributes in common
3: `for j from 0 to m-1:` # iteration over rows of the table
4: `if A & 1 << j:` # if j-th bit of A is 1 (meaning that A contains j-th object)
5: `r = r & rows[j]` # logical "and" between r and j-th row of table
6: `return r`

7: `def Down(B):` # down operator, B is a number representing a subset of attributes.
8: `r = 2**m - 1` # assume that attributes within B have all objects in common
9: `for i from 0 to n-1:` # iteration over columns of the table
10: `if B & 1 << i:` # if i-th bit of B is 1 (meaning that B contains i-th attribute)
11: `r = r & cols[i]` # logical "and" between r and i-th column of table
12: `return r`

13: `def LexicoLessThan(B,B',i):` # checks whether $B <_i B'$ (lexicographic inequality)
14: `if i==0: return True`
whether $B \cap \{0, 1, ...i-1\} = B' \cap \{0, 1, ...i-1\}$
15: `if (B % 2**i) != (B' % 2**i): return False`
if i belongs to B' but not to B
16: `if not (B & 1 << i) and (B' & 1 << i): return True`
17: `else: return False`

18: `def Oplus(B,i):` # for the computation of $B \oplus i$ where B indicates subset of attributes
19: `if i==0 : r = 0`
20: `else: r = B % 2**i` # computing intersection $B \cap \{0, 1, ..., i-1\}$
21: `r = r + 2**i` # union with $\{i\}$
22: `return Up(Down(r))`

23: `def NextIntent(B):` # the next intent that is lexicographically bigger than B
24: `for i from n-1 to 0:`
25: `B' = Oplus(B,i)` # computing $B \oplus i$, starting from biggest i
26: `if LexicoLessThan(B,B',i)` # if $B <_i B'$ then the next intent is B'
27: `return B'`

28: `def NextClosure():` # returns all the intents of a formal context
29: `res = [ ]` # an empty list of intents
30: `Y = 2**m - 1` # corresponding to the set of all attributes
31: `B = Up(Down(0))` # the least intent corresponding to $((\emptyset)^d)^u$
32: `res.Append(B)` # store the least intent
33: `while B != Y do:` # until the current intent is not the set of all attributes
34: `B = NextIntent(B)` # replace the current intent with the next one
35: `res.Append(B)` # store the next intent
36: `endwhile`
37: `return res`

Algorithm 4 Parallelization of the up operator in the binary-based code

```
1: def UpParallel(worker, A): # each worker is assigned to a core
2:   Results = [worker.apply_async(Up, args=(A, RowIndexStart,
     RowIndexEnd)) for core in range(NumberOfCores)] # each worker is re-
     sponsible for processing a chunk of table
3:   Y = 2**n - 1
4:   for res in Results:
5:     Y = Y & res.get() # merging the results obtained by workers
6:   return Y
   ------------------------------------------------------------------
7: with multiprocessing.Pool(NumberOfCores, Func, FuncArg) as workers:
8:   NextClosure(workers) # execute NC using all workers
```

3 Results

This section provides simulation results which consist of three comparisons. We selected them to represent a wide range of tests we conducted, because they depict very well the behavior of all these tests. In section 3.1 we compare the binary-based implementation of NC with the set-based version to verify that the former outperforms the latter specially in cases of larger datasets. We then in section 3.2 investigate the significance of parallel programming in speeding up the NC implementation by comparing the sequential versus the parallelized version of the binary-based approach. Finally in section 3.3 we show that if the dataset is big enough, the time required for executing NC reduces as the number of cores involved in processing increases. For all the executions in this paper, we used the following machine: *1x AMD64 with 8-core Processor @ 3.11 GHz, RAM: 16 GB, OS: Windows 10 Pro.*

3.1 Binary-based versus Set-based computation

We execute NC using two versions of the same algorithm. The first one is based on the set operators and the other one is based on the binary operators. Each dataset is created artificially as a binary table whose elements are independently chosen as 0 or 1 with probability 0.5. The number of attributes in each dataset is set to 15 and the number of objects is varied from 300 to 900. Fig. 1 depicts that the time required for the execution using the binary-based code is less than that in the set-based version. Moreover, the difference between corresponding execution times increases as the number of objects increases. That means the significance of binary computation in reducing required time for the NC execution is highlighted specially for large datasets with big number of rows/columns.

We further compared our binary code with the `concepts` module written in Python that is frequently used for analyzing formal contexts. Table 1 reports the average time spent for executing NC on different-sized datasets. Results are averaged over 5 different simulations. It clearly shows that binary-based approach is much faster than the `concepts` module, even for the small datasets. This

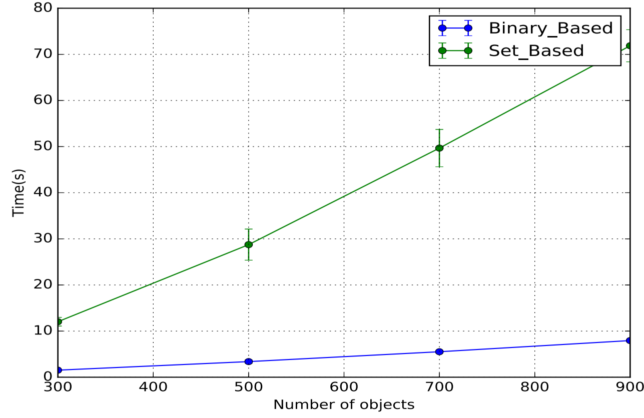

Fig. 1. Binary-based execution (blue curve) of the NC algorithm requires less time (vertical axis) than its set-based version (green curve). The number of attributes in this simulation is set to 15 and the number of objects (horizontal axis) is varied from 300 to 900. Each simulation is repeated 5 times with different datasets to make sure that the result is not biased by the choice of the binary table. The dots on the curves indicate the mean spent time and the error bars indicate the corresponding standard deviation.

numbers should not be considered as an absolute comparison, because both implementations are aiming at very different targets and they must not be realized exactly as the same work. The intention is only to show that our basic implementation is quite efficient, which also underlines the importance of the parallel results, because we compare them with a quite efficient sequential algorithm.

Table 1. Average time required for executing NC using binary-based and python module `concepts`.

Spent time [s] for executing NC		
Table size	Binary-based	`concepts` Module
10 × 5	0.0004	0.0013
20 × 10	0.0055	0.0257
50 × 10	0.0180	0.2309
100 × 10	0.0365	1.0131
200 × 12	0.2316	12.368
300 × 15	1.2937	140.37

3.2 Parallel versus Sequential approach

In this section we investigate whether parallel programming can speed-up the NC implementation and if yes, under which conditions. We parallelized only the *up* operator of the binary code, which was shown (in section 3.1) that it outperforms the set-based version. Therefor, we start from our best sequential algorithm. The goal of the parallel implementation is to use all the available cores of the computing machine to be involved in processing of the *up* operator.

Our expectation was that if the number of rows is sufficiently large, parallelization has a positive effect in reducing the time required for NC executions. This, however, might not be the case if the dataset is too small. The reason is that task distribution between available cores is also time-consuming. Therefore, it might not be reasonable to spend some time for task distribution over available cores when a sequential approach can handle the task itself in a very short amount of time. The same argument can be used for explaining why we did not parallelize the *down* operator since the number of attributes in our examples is not big enough.

We tested our hypothesis by executing the NC algorithm on databases with 15 attributes; but with different number of rows ranging from 3000 to 9000. As depicted in Fig. 2 the average time spent for executing NC using our parallel approach (using all 8 cores) is less time consuming than the best sequential approach, when the number of rows is bigger than roughly 4200. When the number of rows is small, the sequential approach takes less time in average. We can conclude that for relatively small datasets we can start to gain in performance by using parallelization. As this parallelization is very easy to realize and to maintain, this opens some interesting possibilities even for theoretical work.

3.3 Parallel implementation using different number of cores

In order to see how the time required for the NC execution varies as a function of cores involved in processing, we apply NC using different numbers of cores on three databases. All databases have 15 attributes; but with different number of objects. Figure 3 shows that when the number of rows corresponding to the number of objects in a dataset reach a certain level, then the time required for NC reduces as the number of cores increases. This makes sense because parallelization in general speeds up the execution. This statement is, however, not always true. In the case when the number of objects is small the execution time goes up with the number of cores (see the blue curve in Fig. 3). This is because that the task distribution among the cores is time-consuming and this time actually increases with the number of cores.

4 Conclusion

In this paper we proposed to use two techniques that are usually used for speeding up the execution of algorithms: (1) binary computation and (2) parallel

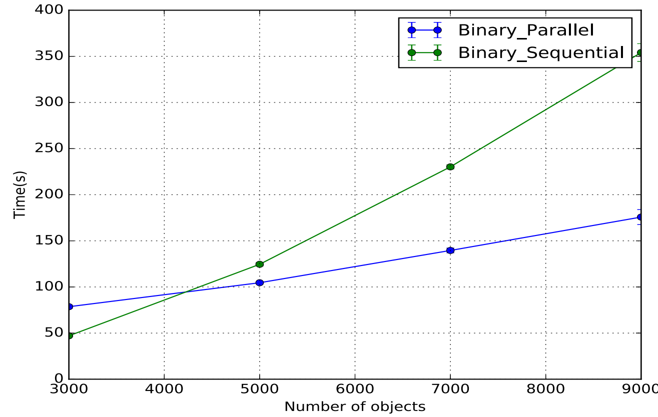

Fig. 2. Parallel versus sequential execution of the NC algorithm. The time (vertical axis) required for the parallel execution of NC (blue curve) is less than the time needed for our best sequential implementation (green curve) if the number of objects is sufficiently big (here, > 4200). The figure depicts the average time spent for executing the NC algorithm on databases with 15 attributes and different number of objects (horizontal axis). Each database is created artificially as a binary table whose elements are independently chosen as 0 and 1 with probability 0.5. Each simulation is repeated for 5 times. Error bars indicate the standard deviation of the corresponding spent times.

programming. We specifically applied these two techniques to the NextClosure algorithm, a fundamental tool for extracting formal concepts from a context. At the same time, we insisted on maintaining implementations that are very close to the formal definitions of these algorithms in order to conduct practical as well as theoretical work. We showed that if datasets are large, both techniques reduce the time required for executing the NC algorithm over the standard algorithms based on sets. We also noticed that both proposed techniques do not necessarily create a speedup when the dataset is not sufficiently large.

The work presented here is the basis for our future projects where we expand our ideas to a wider range of parallel architectures. We would expect that taking advantage of other types processing infrastructures that are readily available can speed up the NC implementation even further and make it interesting for a large audience. We have also some promising results for improving the sequential algorithms as well, which could open the door for FCA to become more widespread also in the context of Big Data.

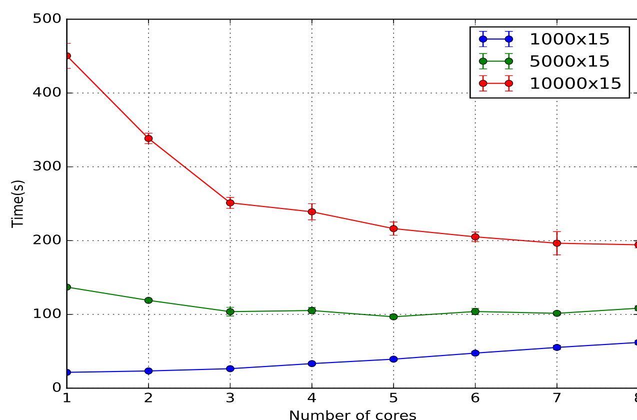

Fig. 3. Parallel execution of the NC algorithm using different number of cores. Three databases with 15 attributes and different number of objects are chosen. Increasing the number of cores reduces the time spent when the number of rows is sufficiently large (red curve corresponding to $10,000$ rows). If the number of rows is not big enough (blue curve with $1,000$ rows), the time spent increases with number of cores. This is because the time required for task distribution over cores itself is not negligible. For the intermediate number of rows (red curve with $5,000$ rows) the spent time lies between the two extremes. Once again each dataset is created artificially as a binary table whose elements are independently chosen as 0 or 1 with probability 0.5. Each simulation is repeated for 5 times. Error bars indicate the standard deviation of the spent time.

References

1. Bernhard Ganter and Rudolf Wille. *Formal concept analysis: mathematical foundations*. Springer Science & Business Media, 2012.
2. Bernhard Ganter. *Two basic algorithms in concept analysis*. Springer, 2010.
3. Radim Belohlavek. Introduction to formal concept analysis. *Palacky University, Department of Computer Science, Olomouc*, 2008.
4. Philipp Cimiano, Andreas Hotho, and Steffen Staab. Learning concept hierarchies from text corpora using formal concept analysis. *J. Artif. Intell. Res.(JAIR)*, 24:305–339, 2005.
5. Paolo Tonella and Mariano Ceccato. Aspect mining through the formal concept analysis of execution traces. In *Reverse Engineering, 2004. Proceedings. 11th Working Conference on*, pages 112–121. IEEE, 2004.
6. Guoqian Jiang, Katsuhiko Ogasawara, Akira Endoh, and Tsunetaro Sakurai. Context-based ontology building support in clinical domains using formal concept analysis. *International journal of medical informatics*, 71(1):71–81, 2003.
7. Lotfi Lakhal and Gerd Stumme. Efficient mining of association rules based on formal concept analysis. In *Formal concept analysis*, pages 180–195. Springer, 2005.
8. Petr Krajca and Vilem Vychodil. Distributed algorithm for computing formal concepts using map-reduce framework. In *Advances in Intelligent Data Analysis VIII*, pages 333–344. Springer, 2009.

9. Biao Xu, Ruairí de Fréin, Eric Robson, and Mícheál Ó Foghlú. Distributed formal concept analysis algorithms based on an iterative mapreduce framework. In *Formal Concept Analysis*, pages 292–308. Springer, 2012.
10. Sergei O Kuznetsov. A fast algorithm for computing all intersections of objects in a finite semi-lattice. *Automatic documentation and Mathematical linguistics*, 27(5):11–21, 1993.
11. Sergei O Kuznetsov. Learning of simple conceptual graphs from positive and negative examples. In *PKDD*, volume 99, pages 384–391. Springer, 1999.
12. Petr Krajca, Jan Outrata, and Vilém Vychodil. Advances in algorithms based on cbo. In *CLA*, pages 325–337, 2010.
13. Simon Andrews. In-close, a fast algorithm for computing formal concepts. 2009.
14. Christian Lindig. Fast concept analysis. *Working with Conceptual Structures-Contributions to ICCS*, 2000:152–161, 2000.
15. Sergei O Kuznetsov and Sergei A Obiedkov. Comparing performance of algorithms for generating concept lattices. *Journal of Experimental & Theoretical Artificial Intelligence*, 14(2-3):189–216, 2002.