Some Programming Optimizations for Computing Formal Concepts

ANDREWS, Simon <http://orcid.org/0000-0003-2094-7456>

Some Programming Optimizations for Computing Formal Concepts

Simon Andrews

Conceptual Structures Research Group
Department of Computing
College of Business, Technology and Engineering
and The Industry and Innovation Research Institute
Sheffield Hallam University, Sheffield, UK
s.andrews@shu.ac.uk

Abstract. This paper describes in detail some optimization approaches taken to improve the efficiency of computing formal concepts. In particular, it describes the use and manipulation of bit-arrays to represent FCA structures and carry out the typical operations undertaken in computing formal concepts, thus providing data structures that are both memory-efficient and time saving. The paper also examines the issues and compromises involved in computing and storing formal concepts, describing a number of data structures that illustrate the classical trade-off between memory footprint and code efficiency. Given that there has been limited publication of these programmatical aspects, these optimizations will be useful to programmers in this area and also to any programmers interested in optimizing software that implements Boolean data structures. The optimizations are shown to significantly increase performance by comparing an unoptimized implementation with the optimized one.

1 Introduction

Although there have been a number of advances and variations of the Close-By-One algorithm [14] for computing formal concepts, including [4, 5, 12, 13], optimization approaches used when implementing such algorithms have not been described in detail. Mathematical and algorithmic aspects of FCA have been well covered, for example in [7, 9, 15], but less attention has been paid to programming. Using a bit-array to represent a formal context has previously been reported [5, 11], but without the implementation details presented here. Providing detailed code for these optimizations will be useful to programmers in this area and also to any programmers interested in optimizing software that implements and manipulates Boolean data structures. Thus, this paper sets out to describe and explain these optimizations, with example code, and to explore the classical efficiency trade-offs between memory and speed in a CbO context. As an example CbO-type algorithm, this paper makes use of In-Close2 as presented in [6]. However, the optimization approaches detailed here should be generalizable for most, if not all, algorithms that compute formal concepts.

The programming language chosen is C++; it is often the language of choice for efficient coding as it facilitates low level programming and its compilers are extremely adept at producing efficient assembler.

2 Formal Concepts

A description of formal concepts [9] begins with a set of objects X and a set of attributes Y. A binary relation $I \subseteq X \times Y$ is called the *formal context*. If $x \in X$ and $y \in Y$ then xIy says that object x has attribute y. For a set of objects $A \subseteq X$, a derivation operator $^{\uparrow}$ is defined to obtain the set of attributes common to the objects in A as follows:

$$A^{\uparrow} := \{\ y \in Y \mid \forall x \in A : xIy\ \}. \tag{1}$$

Similarly, for a set of attributes $B \subseteq Y$, the $^{\downarrow}$ operator is defined to obtain the set of objects common to the attributes in B as follows:

$$B^{\downarrow} := \{\ x \in X \mid \forall y \in B : xIy\ \}. \tag{2}$$

(A, B) is a formal concept *iff* $A^{\uparrow} = B$ and $B^{\downarrow} = A$. The relations $A \times B$ are then a closed set of pairs in I. In other words, a formal concept is a set of attributes and a set of objects such that all of the objects have all of the attributes and there are no other objects that have all of the attributes. Similarly, there are no other attributes that all the objects have. A is called the *extent* of the formal concept and B is called the *intent* of the formal concept.

A formal context is typically represented as a cross table, with crosses indicating binary relations between objects (rows) and attributes (columns). The following is a simple example of a formal context:

	0	1	2	3	4
a	×			×	×
b		×	×	×	×
c	×		×		
d		×	×		×

Formal concepts in a cross table can be visualized as closed rectangles of crosses, where the rows and columns in the rectangle are not necessarily contiguous. The formal concepts in the example context are:

$$\begin{array}{ll} C_1 = (\{a, b, c, d\}, \emptyset) & C_6 = (\{b\}, \{1, 2, 3, 4\}) \\ C_2 = (\{a, c\}, \{0\}) & C_7 = (\{b, d\}, \{1, 2, 4\}) \\ C_3 = (\emptyset, \{0, 1, 2, 3, 4\}) & C_8 = (\{b, c, d\}, \{2\}) \\ C_4 = (\{c\}, \{0, 2\}) & C_9 = (\{a, b\}, \{3, 4\}) \\ C_5 = (\{a\}, \{0, 3, 4\}) & C_{10} = (\{a, b, d\}, \{4\}) \end{array}$$

For readers not familiar with Formal Concept Analysis, further background can be found in [17–19].

3 A Re-Cap of the In-Close2 Algorithm

In-Close2 [6] is a CbO variant that was 'bred' from In-Close [2] and FCbO [13,16] to combine the efficiencies of the partial closure canonicity test of In-Close with the full inheritance of the parent intent achieved by FCbO.

The In-Close2 algorithm is given below, with a line by line explanation, and is invoked with an initial $(A, B) = (X, \emptyset)$ and initial attribute $y = 0$, where A is the extent of a concept, B is the intent and X is a set of objects such that $A \subseteq X$.

In-Close2

`ComputeConceptsFrom(`$(A, B), y$`)`

```
1  for j ← y upto n − 1 do
2      if j ∉ B then
3          C ← A ∩ {j}↓
4          if A = C then
5              B ← B ∪ {j}
6          else
7              if B ∩ Y_j = C^↑j then
8                  PutInQueue(C, j)
9  ProcessConcept((A, B))
10 while GetFromQueue(C, j) do
11     D ← B ∪ {j}
12     ComputeConceptsFrom((C, D), j + 1)
```

Line 1 - Iterate across the context, from starting attribute y up to attribute $n - 1$, where n is the number of attributes.

Line 2 - Skip inherited attributes.

Line 3 - Form an extent, C, by intersecting the current extent, A, with the next column of objects in the context.

Line 4 - If $C = A$, then...

Line 5 - ...add the current attribute j to the current intent being closed, B.

Line 7 - Otherwise, apply the partial-closure canonicity test to C (is this a new extent?). Note that $Y_j = \{y \in Y | y < j\}$. Similarly, $C^{\uparrow_j}$ is C closed up to (but not including) j: $C^{\uparrow_j} = \{\ y \in Y_j \mid \forall x \in C : xIy\ \}$

Line 8 - If the test is passed, place the new (child) extent, C, and the location where it was found, j, in a queue for later processing.

Line 9 - Pass concept (A, B) to notional procedure `ProcessConcept` to process it in some way (for example, storing it in a set of concepts).

Lines 10 - The queue is processed by obtaining each child extent and associated location from the queue.

Line 11 - Each new partial intent, D, inherits all the attributes from its completed parent intent, B, along with the attribute, j, where its extent was found.

Line 12 - Call `ComputeConceptsFrom` to compute child concepts from $j+1$ and to complete the closure of D.

4 Implementation of the Formal Context as a Bit Array

Common to most efficient implementations of CbO-type algorithms is the implementation of the formal context as a bit array, with each bit representing a Boolean 'true/false' cell in the cross-table. Such an approach leads to efficient computation in two ways: it allows for bit-wise operations to be performed over multiple cells in the context at the same time and reducing the size of cells to bits allows a much larger portion of the context to be held in cache memory. So, for example, in a 64 bit architecture the formal context can be declared in the heap using C++ thus:

```
1         unsigned __int64 **context;
```

Once the number of objects, `m`, and number of attributes, `n`, are known, the required memory can be allocated thus:

```
1 /* create empty context */
2 //calculate size for attributes - 1 bit per attribute
3 nArray = (n-1)/64 + 1;
4 //create one dimension of the context
5 context = new unsigned __int64 *[m];
6 for (i = 0;i<m;i++){   //for each object
7    context[i] = new unsigned __int64[nArray];//create a row of
8    for(j=0;j<nArray;j++) context[i][j]=0;    //attributes
9 }
```

Clearly this is a memory efficient way of storing the context: not only are bits being used to represent the individual cells, but dynamic memory allocation is being used to declare only the number of bits required for a particular context. Although dynamic memory allocation is a relatively time-consuming process, here this does not matter as the context is being allocated once only, before the invocation of the main algorithm.

It is important to declare the context so that attributes are contiguous in memory, rather than objects. This structure allows the efficient use of cache memory when the processing is operating on contiguous attributes, such as the iteration across attributes in In-Close. A cache-line will be filled with row of the table, rather than a column, so that contiguous attributes are readily available. Arranging the context column-wise in memory would mean that the subsequent processing would be operating 'against the grain' of memory, causing continuous cache-misses when trying to access contiguous attributes, as the next attribute would be a whole column-worth of memory away from the current one.

Obviously the use of bits to represent the Boolean cells of the cross-table requires careful programming to identify specific cells in the cross-table. Each 64 bit unsigned integer represents 64 cells in a row of the cross-table, thus the arithmetic required is the use of modulo-64 to identify a required cell. For example, attribute 137 would be bit 9 of integer 3: 137 *mod* 64 = 9 and 137 *div* 64 = 2.

If the formal context is being input as a `cxt` file, for example, the program will need to populate the bit array by reading and parsing rows of character strings where '`.`' represents an empty cell and '`X`' represents a cross. For example, a single row in the formal context in a `cxt` file will look something like this:

```
...X.....XX.........X........X...X.X...XX...........X....X.....X
```

The procedure to input the formal context will then look something like this:

```
/* input instances (rows) and translate into temporary context */
  //for each row (object)
  for(i = 0; i < m; i++){
    //get instance
    cxtFile.getline(instance, instanceSize);
    //for each attribute
    for(j = 0;j < n; j++){
    //if object has the attribute
    if(instance[j] == 'X'){
      //set context bit to true at byte: i div 8, bit: i mod 8
      contextTemp[j][(i>>6)] |= (1i64<<(i%64));
      //increment column support (density of Xs) for attribute j
      colSup[j]++;
      }
    }
  }
```

Because binary arithmetic is being used, the C++ bit shift operators `<<` and `>>` provide an efficient means of implementing modulo-64. Thus, `j>>6` shifts the bits in `j` rightwards by 6 bits, which is equivalent to integer division by 64 ($2^6 = 64$). This identifies the required 64 bit integer in the row of the bit array. The *mod* operator in C++ is `%` and the 64 bit literal integer representation of 1 is defined as `1i64` so, similarly, bit shifting `1i64` leftwards by `j%64` places 1 at the required bit position in a 64 bit integer, with all other bits being zero. The C++ bit-wise logical 'or' operator, `|`, can then be used to set the required bit to 1 in the context.

Bit shift operators and bit-wise logical operators are extremely efficient (typically taking only a single CPU clock-cycle to execute) and thus are fundamental to the fast manipulation of structures such as bit arrays. This becomes even more important in the main cycle of CbO-type algorithms and in efficient canonicity testing, as can be seen later in this paper. However, before considering the implementation of the algorithm itself, some consideration needs to be given to the data structures required for the storage and processing of formal concepts.

Note that a temporary bit array, `contextTemp`, is used to initially store the context in column-wise form, because we next wish to physically sort the columns (see Section 4: Physical Sorting of Context Columns, below). After sorting, the context will be translated into row-wise form in the permanent bit-array, `context`, for the main computation.

5 Physical Sorting of Context Columns

It is well-known that sorting context columns in ascending order of support (density of Xs) significantly improves the efficiency of computing formal concepts in CbO-type implementations. The typical approach is to sort pointers to the columns, rather than the columns themselves, as this takes less time. However, in actuality, physically sorting the columns in large formal contexts provides better results, because physical sorting makes more efficient use of cache memory. If data is contiguous in RAM, cache lines will be filled with data that are more likely to be used when carrying out column intersections and when finding an already closed extent in the canonicity test. This can significantly reduce level one data cache misses, particularly when large contexts are being processed [3]. The overhead of physically sorting the context is outweighed by the saving in memory loads.

Thus the column-wise bit array, `contextTemp`, is physically sorted by making use of an array of previously logically sorted column indexes, `colOriginal`:

```
/* rewrite sorted context (physical sort) */
int tempColNums[MAX_COLS];
int rank[MAX_COLS];
  for(j = 0; j < n; j++){
          //use the sorted original col nos to index the physical sort
    tempColNums[j]=colOriginal[j];
    rank[colOriginal[j]]=j; //record the ranking of the column
  }
  for(j = 0; j < n - 1; j++){
    for(i = 0; i < mArray; i++){
      unsigned __int64 temp = contextTemp[j][i];
      contextTemp[j][i] = contextTemp[tempColNums[j]][i];
      contextTemp[tempColNums[j]][i] = temp;
    }
    //make note of where swapped-out col has moved to using its rank
    tempColNums[rank[j]]=tempColNums[j];
    rank[tempColNums[j]]=rank[j];
  }
```

If, for example, `tempColNums = [4,7,0,2,1,6,5,3]`, it means that column 4 is the least dense column and column 3 the most dense. The array `rank` is used to record and maintain the relative ranking of the columns in terms of density. Thus, in this example, `rank = [2,4,3,7,0,6,5,1]`, which means that that column 0 has ranking 2, column 1 has ranking 4, column 2 has ranking 3, and so on.

Once the columns have been physically sorted, the column-wise context, `contextTemp`, is written, a bit at a time, into the row-wise context, `context`, ready for the main computation:

```
for(int i=0;i<m;i++){
  for(int j=0;j<n;j++){
    if(contextTemp[j][(i>>6)]&(1i64<<(i%64)))
      context[i][(j>>6)] = context[i][(j>>6)]|(1i64<<(j%64));
  }
}
```

6 Storing and Processing Formal Concepts

For the purposes of efficiency, it would be desirable to store each formal concept literally and completely, say as a two-dimensional array of objects (for the extents) and a corresponding two-dimensional array of attributes (for the intents). Adding a new formal concept as it is computed would require very little in the way of data management and processing the computed concepts would be via simple iteration of the arrays. However, if we wish to deal with large numbers of objects and attributes, and large numbers of formal concepts, it soon becomes impossible to store them in available memory and in any case would be a very inefficient means of storing them with very little of the allocated memory actually being used (if there are 10,000 objects, for example, each extent would need to be declared as size 10,000 even though most extents will contain far fewer objects). An alternative approach, in an attempt to avoid memory considerations altogether, would be to process each concept as it is computed and not to store them at all. However, it is often the case that the 'processing' is not specified. It may be more useful to compute all formal concepts as a service for later batch-type processing. Or, if the processing is simply to output the concepts to a file, outputting each concept as it is computed is a terribly inefficient process and would make any attempt to optimize the implementation of the algorithm redundant. The approach adopted here, for In-Close2, is a compromise: it is decided to store the formal concepts for later processing, but to store them in such a way as to reduce the memory required, whilst maintaining an efficient means of data management. For speed, the most efficient data structure to use to store intents and extents would be standard two-dimensional arrays, indexing each item in its entirety. However, given that the storage space for each extent would need to be of size m, and size n for each intent, it is quickly apparent that the memory required for a typical computation would be enormous and impractical. Compromises need to be made, and here extents are stored in full but in a list, with the memory required for an extent being the size of the extent. This still often requires a significant amount of memory, typically in the order of several MBs, which, although large, is generally practicable on today's standard PCs. Whilst managing a list requires additional indexing overheads (storing starting points and sizes for each extent) the number of additional computational operations required is not large, enabling extents to be stored for later processing but without drastically impacting speed. This illustrates, in the context of FCA, some typical trade-offs between memory and speed that are so common in dealing with optimization problems.

Another approach would be to employ dynamic memory allocation, allocating memory on the fly (e.g. using `malloc`) with the exact size required for storage. Whilst this intuitively seems like a sensible idea, the process of dynamic memory allocation is rather slow and employing it to store extents and intents increases run-time enormously.

Further computational details (and trade-offs) in the handling of intents and extents are given below.

6.1 Intents

In the In-Close2 algorithm, each child intent inherits fully its parent's attributes. Each child is then specialized by adding one or more new attributes which are then, along with its parental attributes, inherited by the next generation, and so on. Thus, it seems sensible to store the intents in that natural 'tree' structure, thus avoiding repetition of inherited attributes. To further save memory, a single dimensional, linear memory, structure can be used with the addition of meta-data to store the start of each intent in memory, the number of 'own' attributes in each intent (i.e. those attributes in an intent not inherited from its parent) and a pointer to a child intent's parent intent so as to be able to obtain the inherited attributes later when the concepts are 'processed'. The data structure can be declared in C++ as follows:

```
int *B;   //store for intents - memory will be allocated
          //for B at start of main program
int sizeBnode[MAX_CONS]; //number of 'own' attributes in each intent
int *startB[MAX_CONS];    //pointers to start of intents
int nodeParent[MAX_CONS]; //pointers to parent intents
int *bptr; //will point to next available location in B
```

At the start of the main program, the memory allocation for storing intents and pointer initialization is carried out:

```
B = new int[MAX_FOR_B];
bptr = B;
```

Note that there are some predefined literals here, namely `MAX_CONS` (the maximum number of concepts that can be computed) and `MAX_FOR_B` (the amount of memory to be allocated for storing intents). Although it would be possible to dynamically allocate memory on the fly, in the process of computing the concepts, the overheads in time required for this would reduce the efficiency of the implementation. Again, a compromise has been made, in this case to increase efficiency but at the expense of setting arbitrary limits on quantities and sizes.

6.2 Extents

In the In-Close2 algorithm, a new extent is formed as a sub-set of objects from its parent extent. With the breadth then depth approach of the algorithm, it would be possible to avoid repetition of objects (and thus save memory) by removing the child objects from the parent once all the children have been found and then adding meta-data to link the children to the parent. However, this amount of data management would incur significant time overheads, so again a compromise is chosen, in this case to store each extent in full, back-to-back, in memory, adding meta-data to record the starting address of each extent and its size. Thus, although individual objects may be repeated in a number of extents, there is no 'empty', wasted memory that would occur if a two-dimensional array was being used. The data structure can be declared in C++ as follows:

```
1  int* A;   //store for extents - memory will be allocated
2            //at start of main program
3  int* startA[MAX_CONS];   //pointers to start of extents
4  int sizeA[MAX_CONS];     //extent sizes
```

At the start of the program, the memory allocation for storing extents is made, making use of another arbitrary literal for the amount of memory to use:

```
1  A = new int[MAX_FOR_A];
```

Now that the underlying data structures are in place, it is possible to present the optimized implementation of the algorithm itself.

7 Implementation of the Algorithm

The optimized implementation of the InClose2 algorithm is presented below. A key optimization to note is the use of one-dimensional bit arrays (`Bparent` and `Bcurrent`) to represent intents in Boolean form. It was described above how intents are stored in a tree structure to save memory, as opposed to storing each intent separately and in full. Even if they were stored in bit form, the memory required for a large number of intents, each with a maximum size of n would be prohibitive. However, in the algorithm, to skip inherited attributes (line 2) it is necessary to search for j in B. To search the B tree data structure would require some significant time overhead. Searching the intent in Boolean form, however, is simply a logical 'and' bit-wise test (`&` in C++), locating the required bit in the same way as previously described above:

```
24 if(!(Bcurrent[j>>6] & (1i64 << (j % 64))))
```

The memory required for a single Boolean form intent is insignificant. The parent intent is passed down to the child intent at the next level of recursion, and thus the total number of Boolean-form intents being stored (on the stack) at any one time will be dependent on the level of recursion. This is unlikely to cause problems with memory. Thus, the implementation is a two-way compromise in the end: storing intents globally in a memory efficient data structure for later processing, and at the same time storing intents locally in an operationally efficient data structure to reduce computation time. The optimized code for the In-Close2 algorithm is listed below:

```
1  /*OPTIMIZED IMPLEMENTATION OF INCLOSE2 ALGORITHM */
2  void InClose(int c, int y, unsigned __int64 *Bcurrent)
3  // c: concept number, y: attribute number
4  // Bcurrent: the current intent in Boolean form
5  {
6    /* LOCAL VARIABLES */
7    //attributes where new extents are found
8    int Bchildren[MAX_COLS];
9    //the number of new concepts spawned from current one
```

```
10    int numchildren = 0;
11    //the concept no.s of the spawned concepts
12    int Cnums[MAX_COLS];
13    //a child intent in Boolean form
14    unsigned __int64 Bchild[MAX_COLS/64 + 1];
15
16    //calculate the size of current extent
17    int sizeAc = startA[c+1]-startA[c];
18
19    /*********MAIN CYCLE ***********************************/
20    //iterate across attribute columns in the context
21    //forming column intersections with current extent
22    for(int j = y; j < n; j++) {
23       /* if j is not an element of B then */
24       if(!(Bcurrent[j>>6] & (1i64 << (j % 64)))){
25          /* C = A intersect {j}downarrow */
26          //point to start of current extent
27          int *Ac = startA[c];
28          //point to start of new extent
29          int *aptr = startA[highc];
30          //NOTE: highc is maintained globally as
31          //next available concept number
32
33          //iterate through objects in current extent
34          for(int i = sizeAc; i > 0; i--){
35             //looking for them in current attribute column
36             if(context[*Ac][j>>6] & (1i64 << (j % 64))){
37                //if object is found
38                *aptr = *Ac; //add it to new extent
39                aptr++;
40             }
41             Ac++;  //next object
42          }
43
44          //calculate size of new extent
45          int size = aptr - startA[highc];
46
47          /* if A = C then */
48          if(size == sizeAc){
49             //add current attribute to current intent
50             *bptr = j; //in the B tree
51             bptr++;
52             sizeBnode[c]++;
53             //and in the Boolean form of intent
54             Bcurrent[j>>6] = Bcurrent[j>>6] | (1i64 << (j % 64));
55          }
56          else { //size < sizeAc so:
57             //if new extent is canonical
58             if(IsCannonical(j,aptr,Bcurrent)){
59                /* PUT CHILD IN THE QUEUE */
60                //record the attribute where it was found
61                Bchildren[numchildren] = j;
62                //record the new concept number
63                Cnums[numchildren++] = highc;
64                //record the parent concept number
65                nodeParent[highc] = c;
66                //record the start of the new extent in A
67                startA[++highc] = aptr;
68             }
69          }
70       }
71    }
72
73    /* GET CHILDREN FROM THE QUEUE */
74    for(int i = numchildren-1; i >= 0 ; i--){
75      /* D = B U {j} */
76      //inherit attributes
77      memcpy(Bchild,Bcurrent,nArray*8);
```

```
78      //record the start of the child intent in B tree
79      startB[Cnums[i]] = bptr;
80      //add spawning attribute to B tree
81      *bptr = Bchildren[i];
82      bptr++;
83      sizeBnode[Cnums[i]]++;
84      //and to Boolean form of child intent
85      Bchild[Bchildren[i]>>6] =
86       Bchild[Bchildren[i]>>6] | (1i64 << (Bchildren[i] % 64));
87
88      //close the child concept from j+1
89      InClose(Cnums[i], Bchildren[i]+1, Bchild);
90    }
91  }
```

7.1 Optimizing the Canonicity Test

Possibly the greatest time savings can be made in the implementation of the canonicity test. The test is, essentially, a search and is the code most frequently executed in the program. Any performance efficiencies made here will have a significant impact on the overal computation time. The task is to find the candidate new extent in an earlier column in the context cross-table. If it can be found, then the extent is not canonical and thus not new. However, the search must avoid looking in columns representing attributes already in the intent of the current concept - as these will, of course, contain the new extent as a subset. The primary source of efficiency here, is the exploitation of the context as a bit array. Using 64-bits, the search becomes a fine-grained parallelization, searching 64 columns of the context simultaneously. To avoid looking in columns representing attributes already in the intent, a bit-mask can be used to mask out the attributes in question. The mask can be created simply by inverting the Boolean form of the current intent, `Bcurrent`, so that bit positions corresponding to attributes in the current intent become zero and all others are set to 1. The mask is then applied to each object in the new extent (i.e. each corresponding row of the context) using bit-wise 'and'. If the object is present, the corresponding bit in the mask will remain set, if the object is absent the bit will be zeroed.

Further efficiency is possible by stopping the search as soon as the mask becomes completely zeroed - in other words, as soon as it is clear that an object in the extent is not present in any of the columns, it is unnecessary to search for the other objects. The search can then move on the next 64 columns in the context. Conversely, the search can also be stopped as soon as the extent is found: once found it is then not necessary to search any more columns. The combination of fine-grained parallel processing using bit-wise operators and minimizing the amount of searching required, makes the optimized canonicity test extremely efficient. The optimizied canonicity test code is listed below:

```
1  /* OPTIMIZED CANONICITY TEST */
2  bool IsCannonical(int y, int* endAhighc, unsigned __int64 Bcurrent[])
3  /* y: attribute number, endAhighc: points to end of the new extent */
4  /* Bcurrent: the current intent in Boolean form */
5  {
6     /* CREATE BIT MASK FOR SEARCHING */
```

```
    unsigned __int64 Bmask[MAX_COLS/64 + 1];
    int p; //counter for 64 bit segments
    for(p = 0; p < y>>6; p++){
       //invert 64 bit segments of current intent
       Bmask[p]=~Bcurrent[p];
    }
    //invert last 64 bits up to current attribute
    //zeroing any bits after current attribute
    Bmask[p]= ~Bcurrent[p] & ((1i64 << (y % 64))-1);

    /* SEARCH 64 BIT SEGMENTS OF CONTEXT FOR THE EXTENT */
    for(p=0; p <= y>>6; p++){
       int i; //object counter
       //point to start of extent
       int* Ahighc = startA[highc];
       //iterate through objects in new extent
       for(i = endAhighc - Ahighc; i > 0; i--){
          //apply mask to context (testing 64 cells at a time)
          Bmask[p] = Bmask[p] & context[*Ahighc][p];
          //if an object is not found, stop searching this segment
          if(!Bmask[p])break;
          Ahighc++;      //otherwise, next object
       }
       //if extent has been found, it is not canonical
       if(i==0) return(false);
    }
    //if extent has not been found, it is canonical
    return(true);
}
```

8 Evaluation

Evaluation of the optimization was carried out using some standard FCA data sets from the UCI Machine Learning Repository [8] and some artificial data sets. The comparison was between a version of In-Close with and without the optimizations described above, i.e. without a bit-array for the context (an array of type `Bool` is used instead), without the Boolean form of intent being used to skip inherited attributes (the 'tree' of intents is searched instead), and without physical sorting of the context. The difference in performance on the standard data sets, that can be seen in Table 1, is striking, particularly when bearing in mind that the same algorithm is being implemented in both cases - the code optimization is the only difference.

Table 1. UCI data set results (timings in seconds).

Data set	Mushroom	Adult	Internet Ads
$\|X\| \times \|Y\|$	8124×125	48842×96	3279×1565
Density	17.36%	8.24%	0.77%
#Concepts	226,921	1,436,102	16,570
Optimized	0.16	0.83	0.05
Unoptimized	0.47	2.14	0.39

Artificial data sets were used that, although partly randomized, were constrained by properties of real data sets, such as many valued attributes and a fixed number of possible values. The results of the artificial data set experiments are given in Table 2 and, again, show a significant improvement achieved by the optimized implementation.

Table 2. Artificial data set results (timings in seconds).

Data set	M7X10G120K	M10X30G120K	T10I4D100K
$\|X\| \times \|Y\|$	$120,000 \times 70$	$120,000 \times 300$	$100,000 \times 1,000$
Density	10.00%	3.33%	1.01%
#Concepts	1,166,343	4,570,498	2,347,376
Optimized	0.98	8.37	9.10
Unoptimized	2.42	18.65	33.21

9 Conclusions and Further Work

It is clear that certain optimization techniques can make a significant difference to the performance of implementations of CbO-type algorithms. Bit-wise operations and efficient use of cache memory are big factors in this, along with a choice of data structures for storing formal concepts that make a good compromise between size and speed, given the memory typically available and addressable in standard personal computers. Clearly, with more specialized and expensive hardware and with the use of multi-core parallel processing, other significant improvements can be made. However, as far as optimizations are concerned, the ones presented here are probably the most important.

Although space does not permit here, it would be interesting, perhaps in a future, expanded work, to investigate the individual effects of each optimization. It may be that some optimizations are more useful than others. Similarly, it may be interesting to investigate the comparative effectiveness of optimization with respect to varying the number of attributes, number of objects and context density.

The power of 64-bit bit-wise operators naturally leads to the tempting possibility of using even larger bit-strings to further increase the level of fine-grained parallel processing. So-called streaming SIMD extensions (SSEs) and corresponding chip architecture from manufacturers such as Intel and AMD [1, 10] provide the opportunity of 128 and even 256 bit-wise operations. However, our early attempts to leverage this power have not shown any significant speed-up. It seems that the overheads of manipulating the 128/256 bit registers and variables are outweighing the increase in parallelism. It may be because we are currently applying the parallelism to the columns of a formal context (the bit-mask in the canonicity test) rather than the rows, that we are not seeing good results from

SSEs. Whereas there are typically only tens or perhaps hundreds of columns, there are often tens or even hundreds of *thousands* of rows, particularly if we are applying FCA to data sets. Thus, a 256-bit parallel process is likely to have more impact used column-wise than row-wise. The task will be to work out how to incorporate this approach into an implementation.

It may also be worth exploring how the optimizations presented here could be transferred into other popular programming languages, although interpreted languages, such as Python, are clearly not an ideal choice where speed is of the essence. For Java and C#, there appears to be some debate on efficiency compared to C++. It would be interesting to experiment to obtain some empirical evidence.

In-Close is available free and open source on SourceForge[1].

References

1. AMD. *AMD64 Architecture Programmers Manual Volume 6: 128-Bit and 256-Bit XOP, FMA4 and CVT16 Instructions*, May 2009.
2. S. Andrews. In-Close, a fast algorithm for computing formal concepts. In S. Rudolph, F. Dau, and S. O. Kuznetsov, editors, *ICCS 2009*, volume 483 of *CEUR WS*, 2009.
3. S. Andrews. In-Close2, a high performance formal concept miner. In S. Andrews, S. Polovina, R. Hill, and B. Akhgar, editors, *Conceptual Structures for Discovering Knowledge - Proceedings of the 19th International Conference on Conceptual Structures (ICCS)*, pages 50–62. Springer, 2011.
4. S. Andrews. A partial-closure canonicity test to increase the efficiency of CbO-type algorithms. In *Proceedings of the 21st International Conference on Conceptual Structures*, pages 37–50. Springer, 2014.
5. S. Andrews. A best-of-breed approach for designing a fast algorithm for computing fixpoints of Galois connections. *Information Sciences*, 295:633–649, 2015.
6. S. Andrews. Making use of empty intersections to improve the performance of cbo-type algorithms. In *Proceeding of the 14th International Conference on Formal Concept Analysis*, pages 56–71. Springer, 2017.
7. C. Carpineto and G. Romano. *Concept Data Analysis: Theory and Applications*. J. Wiley, 2004.
8. A. Frank and A. Asuncion. UCI machine learning repository: http://archive.ics.uci.edu/ml, 2010.
9. B. Ganter and R. Wille. *Formal Concept Analysis: Mathematical Foundations*. Springer-Verlag, 1998.
10. Intel. Intel Developer Zone, ISA Extensions, `https://software.intel.com/en-us/isa-extensions`, Retreived June 2016.
11. P. Krajca, J. Outrata, and V. Vychodil. Parallel recursive algorithm for FCA. In R. Belohavlek and S.O. Kuznetsov, editors, *Proceedings of Concept Lattices and their Applications*, 2008.
12. P. Krajca, J. Outrata, and V. Vychodil. FCbO program: http://fcalgs.sourceforge.net/, 2012.

[1] In-Close on SourceForge: `https://sourceforge.net/projects/inclose/`

13. P. Krajca, V. Vychodil, and J. Outrata. Advances in algorithms based on CbO. In M. Kryszkiewicz and S. Obiedkov, editors, *CLA 2010*, pages 325–337. University of Sevilla, 2010.
14. S. O. Kuznetsov. A fast algorithm for computing all intersections of objects in a finite semi-lattice. *Nauchno-Tekhnicheskaya Informatsiya, ser. 2*, 27(5):11–21, 1993.
15. S. O. Kuznetsov. Mathematical aspects of concept analysis. *Mathematical Science*, 80(2):1654–1698, 1996.
16. J. Outrata and V. Vychodil. Fast algorithm for computing fixpoints of Galois connections induced by object-attribute relational data. *Information Sciences*, 185(1):114–127, February 2012.
17. U. Priss. Formal concept analysis in information science. *Annual Review of Information Science and Technology (ASIST)*, 40, 2008.
18. R. Wille. *Formal Concept Analysis as Mathematical Theory of Concepts and Concept Hierarchies*, volume 3626 of *LNCS*, pages 1–33. Springer, 2005.
19. K. E. Wolff. A first course in formal concept analysis: How to understand line diagrams. *Advances in Statistical Software*, 4:429–438, 1993.