Parallel Symbolic Relationship-Counting

Andreas C. Doering
IBM Research Laboratory
8803 Rüschlikon, Switzerland
ado@zurich.ibm.com

Abstract:
One of the most basic operations, transforming a relationship into a function that gives the number of fulfilling elements, does not seem to be widely investigated. In this article a new algorithm for this problem is proposed. This algorithm can be implemented using Binary Decision Diagrams. The algorithm transforms a relation given as symbolic expression into a symbolic function, which can be further used, e.g. for finding maxima. The performance of an implementation based on JINC is given for a scalable example problem.

1 Introduction

Next to sets, binary relations are one of the most basic mathematical objects. Given two sets A and B, a relation $\sim$ is defined by a subset $R_\sim$ of $A \times B$, such that

$$a \sim b, a \in A, b \in B \Leftrightarrow (a, b) \in R_\sim.$$

Given a relation on finite sets one of the most basic questions is how many elements are related to each element of A. This can be represented as a function

$$C_\sim : A \to \mathbb{N} : a \mapsto |\{b : a \sim b\}|,$$

called 'counting function' for the relationship in the following. When viewing the relationship as the edges of a directed graph, the function gives the out-degree of each node. When the relationship defines a function from B to A, the transformation counts the pre-images for each function image.

Of course, there is also a counting function with respect to elements of B. Without loss of generality, in this paper only the first version is studied.

As a simple example consider the relation "less than" $a < b$ on numbers $0 \dots m$. For each given a there are $m - a$ numbers greater than a from the given interval. Formally, $C_<(a) = m - a$. In this paper the more general relation $(a\%p) > (b\%q)$ is used as benchmark, which allows scaling the complexity of the function and the number of bits for $A = \{0 \dots 2^N - 1\}$ and $B = \{0 \dots 2^M - 1\}$. $\%$ stands for the modulo operation. For this relation

$$C(a) = max(0, 2^M/q * (q - a\%p)) + max(0, 2^M\%q - (a\%p)).$$

The proposed algorithm can be implemented using Binary Decision Diagrams [MT98]. I am not aware whether there are other representations of Boolean (or more generally finite) functions which allow the same set of operations, including testing for equality, indexing, composition; if there are then the proposed algorithm can be implemented using those representations as well.

The computer algebra system RELVIEW [BN05] is implemented on the basis of Binary Decision Diagrams, but it does not contain the presented function. JINC [OB08] is a BDD library that supports multithreading on a shared-memory system. It was used for the experimental implementation for this paper.

Algorithms to determine the number of fulfilling inputs for a given function are called SAT-COUNT [Thu12]. This is a more restricted problem as it yields a constant for a given function. This problem is equivalent with relationship counting when the set A has only a single element. SAT-COUNT can handle functions in a more general representation, the creation of a BDD for a given input function can be as complex as SAT-COUNT.

BDDs are used for circuit optimization, circuit verification, state space exploration and similar tasks. The proposed algorithm allows more sophisticated tests, and analysis in these domains. The parallel nature of the algorithm gives hope that it can be used also at a large scale, where the representation of A and B requires 100 or more bits.

In the next section two algorithms for determining the counting function are presented.The first version, a recursive algorithm, is simpler to implement and has weaker requirements to the underlying function representation. The second algorithm is based on Binary Decision Diagrams for the basic data type and exploits the representation structure.

In Section 3 some applications for the algorithm are proposed. One of them is the analysis of network topologies [Dör10]. In Section 4 performance results for a C++ implementation based on the BDD library JINC are given. In the outlook further plans for the refinement of the algorithm are given.

2 Relationship-Counting

One basic idea of the algorithm is that counting the number of elements in a subset can be done by summing the characteristic function over all elements of the underlying universe. Consider for example the set of prime numbers, and assume we have given a function $\texttt{isprime} : \mathbb{N} \to \{0,1\}$, then we can determine the number of primes up to a limit l by $\sum_{i=1}^{l} \texttt{isprime}(i)$. Therefore, given a relation $\sim$ by its characteristic function $r : A \times B \to \{0,1\}, r(a,b) = 1 \Leftrightarrow a \sim b$, determining the counting function can be done by summing over all elements of B:

$C_{\sim}(a) = |\{b : a \sim b\}| = \sum_{b \in B} r(a,b)$.

The summing operator can be considered a function of $|B|$ inputs:

$S : \{0,1\}^{|B|}, S(x_0, \ldots, x_{|B|-1}) = \sum_i (x_i)$.

Hence $C_{\sim}(a) = S(r(a,b_0), \ldots, r(a,b_{|B|-1}))$.

Note, that here functions are added, not numbers, i.e. $(f + g) : a \mapsto f(a) + g(a)$. To do this symbolically, a function representation is needed that allows concatenation of two functions, applying partial fixed values to inputs (i.e. concatenation with constant functions) and forming of elementary functions such as addition. The summing function can thus be built from addition functions by concatenation.

For the remainder of the paper it is assumed that natural numbers, including intermediate sums and the result of the counting function are represented as bit vectors with binary encoding ($\sum_i d_i 2^i$). For the representation of elements from A and B, n_a and n_b respectively bits are used. The relation $\sim$ can therefore be represented as a function $\{0,1\}^{n_a+n_b} \to \{0,1\}$. If not all the codes are used to represent elements of A or B, it is assumed that the representation of $\sim$ will always yield 0 when applied to unused codes. Any given representation can be easily modified to fulfil this condition by AND-ing the characteristic functions for the sets A and B to the relation. It is a particular advantage of BDDs that they can handle sparsely represented sets well. For instance permutations, routing structures, and similar objects can be represented with comparably long bit vectors and one-hot encoding with good performance.

The first algorithm is presented for illustrative purposes. It recursively branches on the individual bits `bi` of B. The recursion stops when a function is found that does not depend on b anymore, including constant functions. All the functions found at the bottom of the recursion are collected in a list. Such a recursion on the BDD-representation, limited to certain layers is typical for many BDD algorithms. In particular, this recursion is a part from the fulfilment set counting algorithm found in [MT98]. In a second step an addition tree is built whose leaves are these collected functions. In the simplest form of this algorithm, all functions are added with weight 1, hence the addition tree corresponds to a population count function.

```
collect(f,i)
{
if (f does not depend on bi)
  return {f}
else
  return append(collect(compose(bi,0,f)),
                collect(compose(bi,1,f)))
}
```

`compose(v,e,f)` replaces the input variable v of function f by the expression e. In order to test whether f depends on b or not for a BDD representation the variable order can be used. By arranging the variables for a at the bottom levels and the b-variables at the top, testing the level of the root node of the function f is sufficient and requires only constant time. Other representations, such as polynomials or conjunctive forms might only have a semi-test. This would also be sufficient but could increase the run-time of the algorithm. One improvement of the recursive algorithm is to test, whether the two compositions (compose(bi,0,f) and compose(bi,1,f)) for the recursion parameter represent the same function. In that case the recursion needs to be calculated only once, and the result is returned with weight two, i.e. every list element is extended by a weight, a natural number. This requires of course an equality test on functions, and, again, a semi-test, that can confirm equality but not exclude it, could be applied as well.

The summation step calculates $\sum w_i f_i$ as multi-bit BDD-function, with weights w_i and Boolean functions f_i. The summing is done by constructing a tree from multi-bit adders, where the width of adders grows as necessary from level to level upwards. The algorithm that constructs the adder accounts the sum of weights in the subtrees and can thus determine the required result width. To incorporate the weight w_i for function f_i into the sum on the leaves of the adder tree, a vector of either the constant-zero function or the function itself is formed. If for instance the weight is eleven, the formed vector would be $(f_i, 0, f_i, f_i)$. This is possible because the function is either 0 or 1, so multiplication is identical with the expression:

```
if (fi) then
  return wi
else
 return 0
```

This optimization already contains the idea of the second algorithm. The idea exploits the observation that during the calculation of bitwise projections, frequently the same sub-functions are reached. By collecting the information and refining the function level by level, each sub-function needs to be handled only once. When using BDDs this approach is already known with the only difference being, that BDDs typically do the weight accumulation down to the leaves and do not stop at an intermediate level as is needed here. BDDs can be viewed as representing a dynamic program and the weight calculation corresponds to the well-known dynamic programming algorithm.

The second algorithm consists of two parts, the collection part and the summing part. Both are similar to the recursive algorithm. In the collection part the variable order of the BDD is modified such that variables for b are at the top and the variables for a at the bottom, if needed. Then, the BDD-graph representing the relation is traversed starting from the root level by level and the weights per node are computed. In some BDD libraries the weight can be stored in the graph nodes itself, otherwise a hash table can be used. The root node is assigned weight one. A child node's weight is incremented by weight of the parent multiplied with the power of two of the number of skipped levels, i.e. by the difference of the variable levels of parent and child minus 1. This is because each skipped level corresponds to a input variable that is not relevant for the given situation, and for that reason, every partial assignment for the inputs corresponding to b results in two assignments including the next level variable, one where the variable is one, and another where it is zero. In all well-known BDD implementations this update has constant effort. The terms (the BDD nodes reached which do not depend on b variables) can also be stored in a hash table.

The summing part is identical to the recursive algorithm.

This second algorithm requires the ability to index functions to store them for instance in a hash table or tree.

Both parts of the algorithm exhibit parallelism. Computing a sum of many terms with complex addition steps can naturally be done in parallel by forming an addition tree. Also, the additions itself can be done partially in parallel, for instance by using carry-inputs per digit and composing the digits afterwards or by calculating the prefix sum for the carries in parallel with known methods.

The algorithm can be improved in several ways. One option relates to the addition of weighted vectors on the lowest level of the addition tree. Assuming we add two vectors for functions f and g with weights 5 and 13. This means that we add the vectors $(0, \ldots, 0, f, 0, f)$ and $(0, \ldots 0, g, g, 0, g)$, forming

$$(0, \ldots, 0, g \text{ xor } f\&g, f \text{ xor } g, f\&g, f \text{ xor } g).$$

As can be seen, two terms, $f\&g$ and f xor g can be reused. The caching and reuse of previously computed expressions is part of most BDD implementations. JINC uses a per-thread computed table, so, if the sum of the vector is computed in one thread, the reuse might happen automatically in the BDD functions. Doing it explicitly guarantees the reuse, in case the size of the computed table is not sufficient.

Another idea for improvement is using carry-save adders for the summation tree, as a hardware implementation would do. BDDs are canonical and hence the resulting BDD is determined only by the given relation and not by the algorithm computing it. However, by using carry-save addition one can expect that the intermediate terms will be smaller.

When building the addition tree it is also not clear which sequence is better, first building the weighted addition tree with abstract variables as inputs and then composing the tree with collected functions or building the addition tree directly with the collected terms. Because JINC does not contain a function for vector compose, only the second variant was implemented.

Since addition is commutative and associative, the addition tree can be structured arbitrarily. In order to reduce the complexity of the intermediate operations it might be advantageous to compute more pairwise sums than needed, testing their size and choosing a set of intermediate sums with the smallest size.

3 Applications

Since the described algorithm handles a basic problem, the range of applications is very wide. One application described in [Dör10] is the counting of paths in a network, where routing is constrained by methods for deadlock avoidance or the existence of faults. In this case, one side of the relation covers the start and end points of the path while the other side represents the path of the network as a vector of nodes. The relation is defined by the properties of the network (adjacent nodes in the path have to be connected), and the requirement that the first node of the path is the start and the last node is equivalent to the end point. Further restrictions such as avoiding certain turns or faulty nodes can be added, thereby refining the relation. The proposed algorithm allows finding the number of minimal paths under these restrictions as a function of the start-end node pair. Since the result is a BDD-represented symbolic function, further transformations, such as finding extrema or comparing to the path-counting function for the fault-free network can be carried out in the same framework.

A second type of applications is the evaluation of approximation algorithms, for instance for scheduling. In this case two relations are built, one that describes the set of solutions, and the second representing the set of solutions that a given algorithm can find. BDDs are traditionally used for state space exploration of digital circuits, the set of solutions of an algorithm is a typical outcome of such a process. I am currently working on investigating crossbar scheduling algorithms, such as iSLIP, with this method. It is yet to be seen what size of problem can be covered on typical machines.

A third class of problems deals with configurable circuits. Figures of interest in this context are the set of functions that can be implemented with a given configurable circuit or a minimal configurable circuit that covers a given set of target functions. As an example, consider a configurable random number generator. It consists of a combinational circuit that implements a function with two sets of inputs, one for a uniform random number source and one for the configuration bits. Several classes of circuits can be considered individually, for instance all circuits consisting of three levels of NAND gates. A circuit in such a class is defined by the wiring between the gates which can be represented as a BDD. Doing so results in a relation that combines the wiring, the inputs and the result. With the proposed algorithm the frequency for each output when the random input is varied can be computed as a BDD function. Further processing, including sorting of result vectors, is needed. First experiments have shown that this method works quite efficiently for reasonable circuit sizes.

4 Results

The second algorithm was implemented using JINC on a 64-bit Linux system based on Intel core i5 processors (two cores, two threads each). The tests were repeated on a Freescale T4240-based system called RDB which provides 24 processor cores, and the results were comparable. Table 1 lists the run time of the two parts of the algorithm for several example problems.

As can be seen, scalability works well, and is better for larger problems. Note, that JINC needs internal locking for shared data structures, in particular the so called “unique” table, with one table per variable. Scaling of part 1 is limited by the global lock for the hash table that collects the individual functions. This can be improved by using a hash-map implementation that allows concurrent insertion, as is provided by Intel’s Threading Building Blocks library. However, the thread management of the TBB is somewhat different than that of the boost library. Since a concurrent hash table is in preparation for the boost library, this improvement was postponed.

Table 1: Results for some example runs, note that 1536=3*512 resulting in a particular simple BDD for the modulo operation. p1 and p2 are the runtimes of the two parts of the second algorithm in seconds.

N	M	p	q	threads	p1	p2	terms
16	16	1697	1879	1	0.05	33	2047
16	16	1536	1879	1	0.04	1.75	2047
16	16	1697	1536	1	0.03	7.3	2047
16	22	10697	1035641	1	0.96	42.5	16383
22	16	1035641	10697	1	0.15	42.5	16384
16	16	1697	1879	2	0.07	18	2047
16	16	1536	1879	2	0.1	1.0	2047
16	16	1697	1536	2	0.06	11.2	2047
16	22	10697	1035641	2	2.08	29.7	16383
22	16	1035641	10697	2	0.3	37.7	16384
16	16	1697	1879	3	0.09	15.7	2047
16	16	1536	1879	3	0.18	0.93	2047
16	16	1697	1536	3	0.08	11.5	2047
16	22	10697	1035641	3	2.5	26.8	16383
22	16	1035641	10697	3	0.4	30.3	16384

5 Outlook

It has to be noted that the chosen example has some particular properties, as can be seen from the number of terms. Another property that was observed is that there are few edges that skip one or several levels. This is a property of the BDD representing the input relation. These level-skipping edges however contribute to the acceleration of the algorithm. More level-skipping edges improve the parallel performance of the first step because the locking for weight updates are better distributed; the current implementation uses one lock per level.

Therefore, one of the most important next steps is the use of other examples, in the best case by applying the algorithm to applications or by integration into a more general tool, such as RELVIEW.

Furthermore, using a concurrent hash table once it is available in the boost library should improve the scalability of the first step of the algorithm.

References

[BN05] Rudolf Berghammer and Frank Neumann. RelView - An OBDD-Based Computer Algebra System for Relations. In Victor G. Ganzha, Ernst W. Mayr, and Evgenii V. Vorozhtsov, editors, *CASC*, volume 3718 of *Lecture Notes in Computer Science*, pages 40–51. Springer, 2005.

[Dör10] Andreas C. Döring. Analysis of Network Topologies and Fault-Tolerant Routing Algorithms using Binary Decision Diagrams. *Parallel and Distributed Processing Workshops and PhD Forum, 2011 IEEE International Symposium on*, 0:1–5, 2010.

[MT98] Christoph Meinel and Thorsten Theobald. *Algorithms and Data Structures in VLSI Design*. Springer-Verlag New York, Inc., 1998.

[OB08] Jörn Ossowski and Christel Baier. A uniform framework for weighted decision diagrams and its implementation. *Int. J. Softw. Tools Technol. Transf.*, 10:425–441, September 2008.

[Thu12] Marc Thurley. An Approximation Algorithm for #k-SAT. In Christoph Dürr and Thomas Wilke, editors, *STACS*, volume 14 of *LIPIcs*, pages 78–87. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2012.