

Compressing Binary Decision Diagrams

Esben Rune Hansen¹ and S. Srinivasa Rao² and Peter Tiedemann³

Abstract. The paper introduces a new technique for compressing Binary Decision Diagrams in those cases where random access is not required. Using this technique, compression and decompression can be done in linear time in the size of the BDD and compression will in many cases reduce the size of the BDD to 1-2 bits per node.

Empirical results for our compression technique are presented, including comparisons with previously introduced techniques, showing that the new technique dominate on all tested instances.

1 Introduction

In this paper we introduce a technique for compressing binary decision diagrams for those cases where random access to the compressed representation is not needed. The two primary areas in which decision diagrams are used in practice are verification and configuration. In both of these areas it is sometimes important to store binary decision diagrams using as little space as possible but without the need for random access. Primarily the need for such compression arises when it is necessary to transmit binary decision diagrams across communication channels with limited bandwidth. In the area of verification this need arises for example when using a networked cluster of computers to perform a distributed compilation of a binary decision diagram [1]. A similar exchange of BDD data takes place in *distributed configuration* as described in [11]. In such approaches the fact that the network bandwidth is much lower than the memory bandwidth can become a major bottleneck as computers stall waiting to receive data to process. Transmitting the binary decision diagrams in a compressed representation can help alleviate this problem. A full version of this paper is available at [4].

Related work The only previous work we are aware of for compressing BDDs for offline storage is the work by Starkey and Bryant[9] and by Mateu and Prades-Nebot[7] which describes techniques for image compression using BDDs. The latter includes a non-trivial encoding algorithm for storing the BDD. Kieffer et.al[5] gives theoretical results for using BDDs for general data compression including a technique for storing BDDs.

Preliminaries For a definition of BDDs please see [2]. We denote a given BDD as $G(V, E)$ and use E_{low} and E_{high} to denote the set of low and high edges respectively. We use $l(u)$ to denote the layer in which a node u is located. An edge (u, v) such that $l(u) + 1 < l(v)$ is called a long edge and is said to skip layer $l(u) + 1$ to $l(v) - 1$. The length of an edge (u, v) is defined as $l(v) - l(u)$. A layer ordering $id_l : V \rightarrow \{1, \dots, |V|\}$ of the nodes in a layered DAG $G(V, E)$ rooted in r is the ordering of V layer by layer in increasing order of

the layer. Nodes at the same layer are ordered as they are visited by a DFS in the DAG starting at r and traversing left edges prior to right edges. We refer to $id_b(v)$ and $id_l(v)$ as “the BFS id of v ” and “the layer id of v ” respectively.

Lemma 1. *Every binary tree can be unambiguously encoded using 2 bits pr. node.*

To achieve such an encoding each node v is encoded using two bits. The first bit and the second bit is true iff v contains a left and a right child respectively. In order to make decoding possible the order in which the children of already decoded nodes appear in the encoded data must be known.

2 The Compression technique

Our compression technique can be summarized by the following steps:

1. Build a spanning tree on the BDD (Section 2.1).
2. Encode edges in the spanning tree, using Lemma 1.
3. Encode by one bit the order in which the two terminals appear in the spanning tree.
4. Encode the length of the edges in the spanning tree where necessary (Section 2.1).
5. Encode the edges that are not in the spanning tree (Section 2.2).
6. Compress the resulting data using standard compression techniques.

2.1 The spanning tree

We will construct a spanning tree with a minimum number of long edges. For each node v in the BDD with parents u_1, \dots, u_k , we add the edge (u_j, v) that minimizes $l(v) - l(u_j)$ to the spanning tree. This ensures a spanning tree with a minimal number of long edges. In the following, an edge is called a *tree edge* if it is contained in the spanning tree and a *nontree edge* otherwise.

Encoding the lengths of the tree edges The spanning tree is stored as a binary tree where all edges have the same length. Since some of the edges in the spanning tree may correspond to long edges in the BDD, the binary tree itself is not sufficient to reconstruct the layer information during decoding. We therefore encode the location and the length of each long edge that is included in the spanning tree. The location of a long edge (u, v) is uniquely specified by the BFS order of the end point of the edge, that is $id_b(v)$. To encode location of the long edges $(u_1, v_1), \dots, (u_k, v_k)$ we, output a bitvector of length $|V|$ for which entries $id_b(v_1), \dots, id_b(v_k)$ are true and all other entries are false.

¹ IT-University of Copenhagen

² MADALGO, Aarhus University, Denmark

³ IT-University of Copenhagen

2.2 Encoding nontree edges

When the spanning tree and the layer information is encoded, we only need to encode the nontree edges, that is, those edges in the BDD that are not contained in the spanning tree. It is easy to see that there is $|E|/2 + 1$ tree edges (when $|V| > 3$), leaving $|E|/2 - 1$ nontree edges. With access to the spanning tree with restored layer information, the fact that every BDD node except the terminals has two children, the starting point of the nontree edges is known. The end-point of a nontree edge is called an *incomplete child*. We define S as the sequence of incomplete children appearing in layer order of their parent and $id_l(S)$ as the corresponding sequence of layer ids. Below we describe three encodings of nontree edges which combine to encode all the nontree edges.

Incomplete children with large in-degree Standard compression techniques excel at compressing sequences with high redundancy. We note that nodes with in-degree d will appear $d - 1$ times in the sequence of nontree edges. Hence standard compression will efficiently compress those nontree children that have a high in-degree if they are separated from the nodes that have a low in-degree. We split S into two disjoint subsequences H and L , the first containing those incomplete children that have an in-degree larger than a specified threshold, the latter containing the rest. Based on H we construct the sequence of integers S^H on the sequence of nodes $v_1, \dots, v_{|V|}$ in S by encoding $v_i \in H$ as $id_l(v_i)$ and $v_i \in L$ as 0. By 0s we indicate the incomplete children that are not among the incomplete children with high in-degree. The remaining incomplete children L , we code separately, as described in the next two sections.

Incomplete children with small in-degree To encode L we will exploit the fact that the sequence of integers in $id_l(L)$ will in most instances tend to be increasing (this behavior is analysed in more detail in the full version [4]). We exploit this fact by encoding the sequence $id_l(L)$ using delta coding:

Definition 2 (Delta Coding). *Consider any sequence of integers $(i_1, \dots, i_k) \in \mathbb{Z}^k$ for any $k \in \mathbb{N}$. We define the delta coding of (i_1, \dots, i_k) by $\Delta(i_1, \dots, i_k) = (i_1, i_2 - i_1, i_3 - i_2, \dots, i_k - i_{k-1})$*

Long forward edges A nontree edge (u, v) is a *forward edge* if u is an ancestor of v in the spanning tree. Any forward edge (u, v) in the graph with length k can be unambiguously decoded from $id_l(v)$ and k . We label each node v with the number of long-edges that ends in v . We then write the length of the long edges, ordered by their end-points. We introduce a threshold on the number of long forward edges to control the use of this approach. If the threshold is not exceeded all long forward edges are instead encoded as described above.

3 Experiments

In this section we provide empirical results from compressing a large set of BDDs from various sources using the new encoder described in this paper and the encoders from [7] and [5]. We also provide results for a naive encoder, which outputs the size of each layer followed by a list of children. Many of the instances we show results for are taken from the configuration library CLib [10]. We apply LZMA[8] to the output of all encoders to produce the final encoding. The Java source code used for these experiments (including a command-line encoder and decoder for BDDs in the BuDDy [6] file format) will be made available along with all instances used in these experiments at [3].

Conclusion From the empirical results (Figure 1) we can see that the naive encoding, being only compressed by LZMA, is outperformed with a factor of up to 20. We also note that the new encoder is consistently able to perform as well or better than the other encoders on all tested instances. In particular the largest BDD in our test (“complex-P3”) required about twice as much space when using either of the two other dedicated encoders.

Name	V	this paper	[7]	[5]	Naive
Product Configuration					
renault	455798	0,90	126%	103%	402 %
renault-dir	1392863	0,23	198%	214%	1352%
pc-CP	16496	0,76	220%	209%	788 %
pc	3467	2,19	224%	211%	436 %
Big-PC	356696	0,38	334%	266%	1345%
Big-PC-dir	1291600	0,17	260%	260%	2035%
Power Supply Restoration					
complex-P3	2812872	0,44	243%	202%	951 %
complex-P2	163432	1,16	181%	167%	541 %
1-6+22-32	20937	1,89	136%	154%	413 %
1-6+22-32-dir	61944	0,99	135%	161%	606 %
Fault Trees					
isp9607	228706	0,63	389%	204%	873 %
isp9605	4570	3,30	130%	145%	305 %
chinese	3590	2,06	214%	160%	450 %
Combinatorial					
5x27queens	562764	4,33	108%	109%	204 %
13x13rook	76808	3,56	210%	165%	311 %
8x8rook	1339	6,03	140%	139%	277 %
8x8queen-dir	2453	2,17	115%	178%	374 %
8x8queen	879	4,29	114%	138%	332 %
Multipliers					
mult-mix-10	42468	9,92*	114%	107%	169 %
mult-apart-10	31260	8,07*	120%	124%	202 %

Figure 1. Above is shown the name and nodecount of each of the instances tested. The result of the new encoder, in bits per node, is then showed, followed by the relative results of the rest of the encoders. The * indicates that delta-coding was not used.

References

- [1] P. Arunachalam, C. Chase, and D. Moundanos, ‘Distributed binary decision diagrams for verification of large circuits’, *ICCD*, **00**, 365, (1996).
- [2] Randal E. Bryant, ‘Graph-based algorithms for boolean function manipulation’, *IEEE Transactions on Computers*, **35**(8), 677–691, (1986).
- [3] Esben Rune Hansen, Srinivasa Rao, and Peter Tiedemann, ‘Bdd compression’. <http://bddcompression.sourceforge.net>.
- [4] Esben Rune Hansen, Srinivasa Rao, and Peter Tiedemann. Compressing binary decision diagrams, 2008. <http://arxiv.org/abs/0805.3267v1>.
- [5] J. Kieffer, P. Flajolet, and E h. Yang, ‘Universal lossless data compression via binary decision diagrams’, in *Proceedings of ISIT 2000*, (2000).
- [6] J. Lind-Nielsen, ‘BuDDy - A Binary Decision Diagram Package’. <http://sourceforge.net/projects/buddy>, online.
- [7] P. Mateu-Villarroya and J. Prades-Nebot, ‘Lossless image compression using ordered binary-decision diagrams’, *Electronic Letters*, **37**, 162–163, (2001).
- [8] Igor Pavlov. 7z lzma sdk. <http://www.7-zip.org/sdk.html>.
- [9] M. Starkey and R. Bryant. Using ordered binary-decision diagrams for compressing images and image sequences, 1995.
- [10] Sathiamoorthy Subbarayan. Clib: configuration benchmarks library. <http://www.itu.dk/research/cla/externals/clib>.
- [11] Peter Tiedemann, Tarik Hadzic, Stuart Henney, and Henrik Reif Andersen, ‘Interactive distributed configuration’, in *Proceedings of CP2006*, pp. 761–765. Springer-Verlag Berlin Heidelberg, (2006).