An Approach to Graph and Netlist Compression

Jeehong Yang, Serap A. Savari
EECS Department
University of Michigan
Ann Arbor, MI 48109, USA
Email: {xosh,savari}@eecs.umich.edu

Oskar Mencer
Department of Computing
Imperial College London
London SW7 2BZ, UK
Email: o.mencer@imperial.ac.uk

Abstract

We introduce an EDIF netlist graph algorithm which is lossy with respect to the original byte stream but lossless in terms of the circuit information it contains based on a graph mining tool SUBDUE. Our algorithm, *CEDIF* (Compressed EDIF), compresses the EDIF file about $2-3\%$ more than the state-of-the-art PAQ text compression algorithm. We also developed a heuristic partitioning algorithm that tries to avoid hurting frequent subgraph patterns by the partitioning process to speed up the graph mining process, and introduced a way to use graph mining tools as a graph compressor so that the compressed graph could be decompressed.

I. Introduction

As VLSI (Very Large Scale Integration) technologies develop, it is possible to increase the density of transistors in the same chip area. This potentially enables the design of more complex circuits. As a circuit becomes more complex, there are more elements to connect, and hence the size of the connection information, called a *netlist*, grows. Circuits in today's technology have netlist files sometimes being described with several hundreds of megabytes, and we anticipate that in the near future the standard format for netlists will result in files requiring several gigabytes each. Therefore, it is important to reduce the size of a netlist.

There are two approaches to compressing a netlist. Since netlist files are usually structured texts which have some similarities with XML (eXtended Markup Language), it is possible to directly compress a netlist with a structured text compression algorithm such as LZCS[1], or to first convert it into an XML file and then apply various XML compression algorithms such as XMill[7], XML-PPM [2], and XComp[6]. These methods offer some compression with relatively low computational complexity. However, since these compression algorithms focus on compressing the structured text itself and ignore the connection information it contains, it is natural to explore alternative approaches based on compressing the labeled graph corresponding to the connection information.

Research in *graph mining* offers tools and techniques to discover important subgraph structures from input graph(s). There are two types of graph mining problems. One of them seeks the important subgraph structures from a set of graphs $\{G_1, \cdots, G_n\}$[13], [11], [12]. For this case, most algorithms determine if a subgraph pattern is in G_i or not, and hence, any subgraph could be counted at most n times. Such algorithms are not adequate by themselves for compressing a single graph. The primary application of the tools is in the study of molecular physics, and the input graph sets are usually parts of some molecule. The other graph mining literature aims to uncover the important subgraph

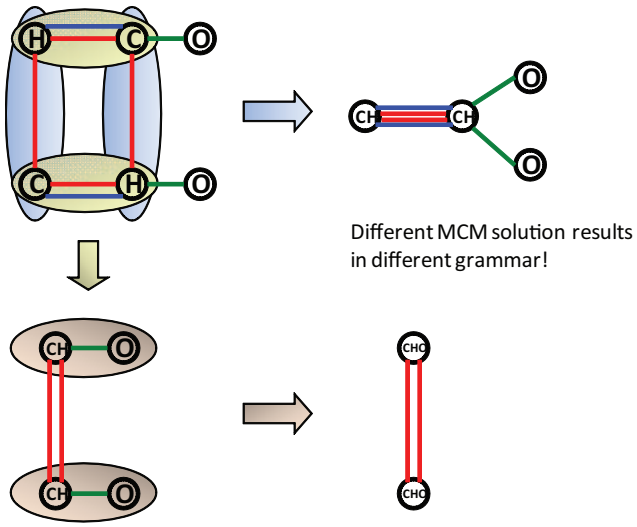


Fig. 1. An example circuit which can have different graph grammars by choosing different maximum cardinality matching solution for GRAPHITOUR.

structures from a single input graph $G = (N, E)$[19], [10], [15]. For this case, a specific subgraph pattern could occur more than one time and up to n times for the input graph. Although the discovered subgraph structures could be used toward graph compression, most of the literature in graph mining only focuses on improving the understanding of graph structures.

Among the graph mining tools, SUBDUE[10] and GRAPHTOUR[15] discusses graph compression. SUBDUE uses a heuristic algorithm to discover frequent subgraphs, and it applies a form of MDL (Minimum Description Length) to decide in a greedy fashion what is the most effective subgraph pattern at any step for compressing the graph. However, the shortcomings of SUBDUE are that it does not offer a compression method that is decompressible, it does not discover all the important subgraphs and it requires massive computation[19]. GRAPHITOUR considers the most frequent edges at a time, and contracts parts of them which satisfy the MCM (Maximum Cardinality Matching) problem[14]. However, since it is only considering an edge type at a time, it lacks in structural view, and hence, might miss some important subgraph structures for compression. For example, in Fig.1, MCM solution for a subgraph induced by each edge type (C−H, C−O) has two edges at most. So, we can choose any MCM solution for GRAPHITOUR. Consider we chose a MCM solution from the C−H edge. Even though, the top/bottom pair solution and the left/right pair solution results in different compressed graph, GRAPHITOUR algorithm does not specify which one to choose, and hence, can choose the worse one.

There also has been research on efficient ways to represent a graph [21], [23]. The *graph representation* literature has focused on what is the efficient way to represent the adjacency list (or the adjacency matrix) of the graph. They consider graph structures such as graphs of bounded genus[17] and planar graphs[22] to obtain more tight representation bounds, but does not consider repeated structures within the graphs.

In this paper, we describe an EDIF netlist compression algorithm based on a labeled graph compression technique, and

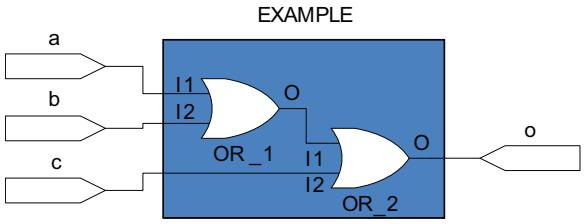


Fig. 2. The circuit `EXAMPLE: o = (a OR b) OR c`

- we develop a graph partitioning algorithm which is useful for preserving frequent subgraph patterns of the graph,
- we modify the state-of-the-art graph mining tool so that it can compress the graph and the graph can be recovered from the compressed file,
- we compare our results with other compression algorithms that can compress EDIF netlist files.

In Section II, we briefly review the structure of EDIF netlist files. In Section III, we discuss the EDIF netlist compression algorithm based on graph compression techniques. We provide experimental results in Section IV, and conclude in Section V.

II. The EDIF Netlist

There are many ways to describe a circuit netlist and several netlist description formats. In this paper, we only concentrate on EDIF (Electronic Data Interchange Format)[8] because most other formats can only be read into a specific type of CAD (Computer Aided Design) tool. EDIF was proposed back in the 1980s to transmit circuit information to various CAD tools, and it is now widely supported by most vendors of CAD tools. EDIF could be used to describe more than just circuit netlists, but we focus here only on circuit netlists.

Before describing the format of EDIF files, we explain some terms that are standard in discussing the structure of a circuit. (See Fig. 2 for an example circuit.)

- Cell : A *cell* is a basic block for describing a circuit. Each cell has *ports* from which it receives or transmits an electric signal. To describe a circuit efficiently, we use a set of cells called a *standard cell library*. A standard cell library contains a cell list, the information on how the cells are implemented in the physical domain, and the properties of each cell such as its function, size, delay, and power consumption. In Fig. 2, the circuit consists of one cell, the `OR` gate which has two input ports `I1`, `I2`, and one output port `O`.
- Instance: An *instance* is an embedding of a cell in the circuit and is used to distinguish the cells that are used multiple times. In our example, we have two instances `OR_1` and `OR_2` using the same cell, the OR gate.
- Net: a *net* describes how the ports are connected within the instances. In Fig. 2, we have five nets: (`EXAMPLE.a`, `OR_1.I1`), (`EXAMPLE.b`, `OR_1.I2`), (`OR_1.O`, `OR_2.I1`), (`EXAMPLE.c`, `OR_2.I2`), and (`OR_2.O`, `EXAMPLE.o`), where (`A.a`, `B.b`) denotes there is a connection from port `a` of `A` to port `b` of `B`.

A. Format

An EDIF file is a text file having a grammar like the programming language LISP. The basic EDIF syntax is called a *construct*. A construct begins with an opening parenthesis

```
(cell OR (cellType GENERIC)
 (view Netlist_representation (viewType NETLIST )
  (interface
   (port I1 (direction INPUT))
   (port I2 (direction INPUT))
   (port O (direction OUTPUT))
  )))
……
(instance OR_1
 (viewRef Netlist_representation
  (cellRef OR(libraryRef my_class))
 ))
……
(net a
 (joined
  (portRef a)
  (portRef I1 (instanceRef OR_1))
 ))
……
```

Fig. 3. Part of EDIF netlist describing `EXAMPLE` circuit

'(' and a *tag*; this is followed by a list of items ending with a closing parenthesis ')'. Those items may be *elements* consisting of data items, or they may be other constructs which build a nested structure[8].

In an EDIF file, the construct describes the cells, instances, and nets of a circuit. Fig. 3 shows part of an EDIF file which describes the `EXAMPLE` circuit in Fig. 2. The first block of Fig.3 defines the `OR` cell and its ports, the second block defines the `OR_1` instance, and the last block defines the net (`EXAMPLE.a`, `OR_1.I1`). Note that we used italics to emphasis the tags of each construct in Fig.3.

B. Isomorphism

Our focus is on the compression of a netlist graph instead of on structured text compression, and the file we obtain after a compression-decompression process generally differs from the input EDIF file. We say the files are equivalent if the circuits they produce are *isomorphic*. In other words, our compression algorithm is lossy with respect to the original byte stream but lossless in terms of the circuit structure it contains.

The following parts of netlists can be ignored when we concentrate on circuit isomorphism.

1) Description order: It is acceptable to change the element order of cells (or their ports), instances, and nets.
2) Redundant cells: It is acceptable to ignore cell definitions for the cells that are not used in any of the instances because this is redundant information.
3) Instance names: Most of the CAD tools writing EDIF files generate a collection of instance names without a specific importance, and the end-user does not need the original instance names.

III. Netlist Compression Algorithm

In this section, we describe the netlist compression algorithm. The overall process is shown in Fig.4. We start the netlist compression by extracting the connection information in a *graphical* structure from the netlist. We compress the graph structure by discovering

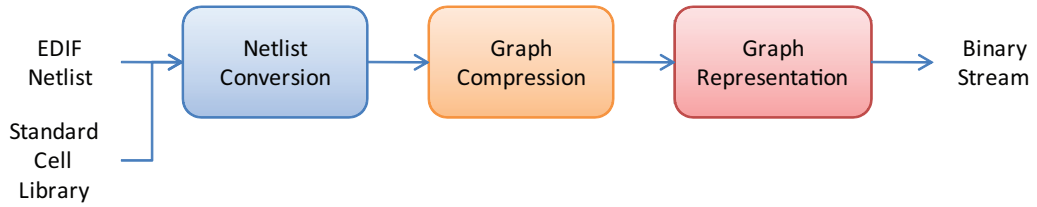
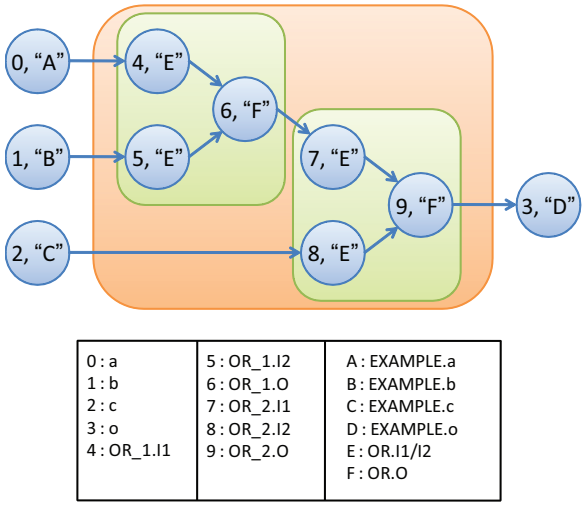


Fig. 4. The netlist compression algorithm



0 : a 1 : b 2 : c 3 : o 4 : OR_1.I1	5 : OR_1.I2 6 : OR_1.O 7 : OR_2.I1 8 : OR_2.I2 9 : OR_2.O	A : EXAMPLE.a B : EXAMPLE.b C : EXAMPLE.c D : EXAMPLE.o E : OR.I1/I2 F : OR.O

Fig. 5. Netlist to Graph conversion of the `EXAMPLE` circuit.

frequent subgraphs. Finally, we represent the compressed graph structure in a binary stream.

A. Netlist Conversion

The graph that we extract from a netlist file should contain all of the connection information. Moreover, the graph should have structures that correspond to the structures of the netlist; i.e., the graph should contain the cell, instance, and net information.

The *nodes* of the graph represent instance-ports, and the *edges* capture the net definition. We will use *node labels* to identify the cell-port for the corresponding instance-port. Finally, the graph is *directed* because the net connections have directions.

Converting a netlist file to a graph is simple. Generate nodes for each instance-port in the circuit, and label each one according to its cell-port attributes. For each instance, connect each input port to its corresponding output port(s). Next, connect all of the nodes that are in the nets with consideration of the port property so that input signals are mapped to input ports and output signals correspond to output ports.

Fig.5 shows the graph corresponding to the netlist in Fig.2. In each node, the first number refers to the node name and the second component provides the node label. The table in Fig.5 shows what the node names and node labels actually mean from Fig.2

B. Graph Compression

The overall graph compression process is shown in Fig.6. We start by partitioning the graph because discovering frequent patterns from a single large graph is computationally

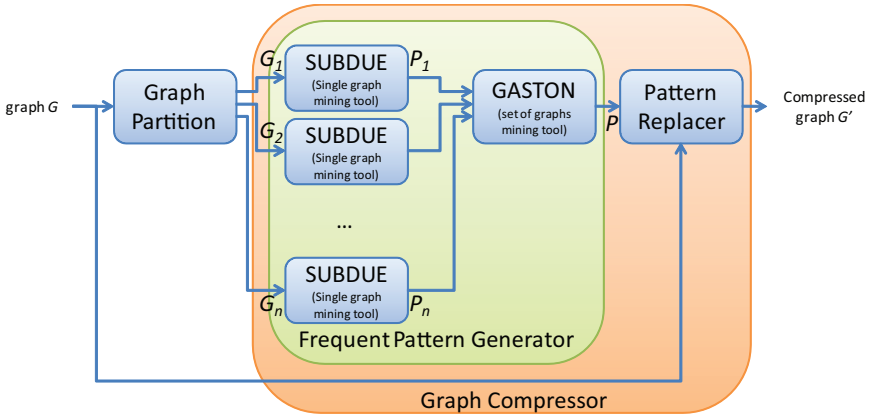


Fig. 6. The graph compression algorithm

expensive. After graph G is partitioned into $\{G_1, \cdots, G_n\}$, we run SUBDUE to find the set of subgraphs P_i that can compress the partition G_i, where $i = 1, \cdots, n$. Since there are possible similarities within $\{P_1, \cdots, P_n\}$, we run another graph mining tool, GASTON, to solve a graph mining problem for a set of graphs to find the distribution of frequencies in the SUBDUE-discovered patterns. Next, we compress the graph G by contracting subgraphs which match patterns in P. During this process, we make sure the compressed graph G' can regenerate the original graph G.

1) Graph Partition: The bottleneck in graph compression is in the graph mining step because discovering important subgraph structures requires a search space exponential in the size of the input graph size. For example, we use SUBDUE[10] as our main single graph mining tool. When we run SUBDUE to discover important subgraph patterns among the benchmark circuits, just a few files were processed within a reasonable amount of time. For all of the other files, we stopped the graph mining process after several hours with no result. Therefore, in order to achieve scalability, it is important to partition the graph.

Since we need the original graph back from the partitioned graph, we say that $\{G_1, \cdots, G_n\}$ is a *partition* of graph $G = (N, E)$ if it satisfies the following properties:

- $G_i = (N_i, E_i)$ is a induced subgraph of G for $i = 1, \cdots, n$.
- $\cup_{i=1}^{n} N_i = N$.
- $\cup_{i=1}^{n} E_i = E$ where $\cap_{i=1}^{n} E_i = \phi$.

By, the second and third properties, we could always reconstruct the original graph G from the partition just by using the union operation.

Since the partitioning process might affect the discovered frequent subgraph patterns, we do not wish to partition the graph just by discovering the minimum cuts as is done in [18]. We instead seek to the edges that minimize the effect on the frequent subgraph patterns and gather them until they can serve as cuts.

The following algorithm partitions the graph G into $P = \{G_1, \cdots, G_n\}$ such that $\texttt{sizeof}(G_i) \leq \beta$, where $i = 1, \cdots, n$. First, we initialize the partition P by inserting G; $G_1 = G$. Then, we analyze P to obtain the maximum partition size m and the corresponding part i. If $m \leq \beta$, then we stop partitioning the graph. Otherwise, we iterate the following procedures until $m \leq \beta$. In each iteration, we first analyze the edge

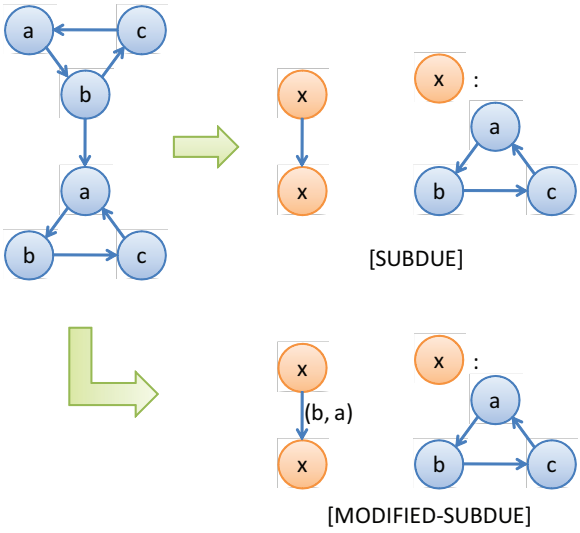


Fig. 7. Decompressible Graph Compression

label[1] frequency. Then, we move the edges that occur less than α times each from G_i to a partition G_k, where $\texttt{sizeof}(G_k) + \texttt{sizeof}(\texttt{Nonfrequent\_Edges}) \leq \beta$. Next, we group the edges having the most frequent edge label meaning we are treating it as a node. We change the label of each edge which is an incoming/outgoing edge of an edge having the most frequent edge label in order to we consider the higher level edge frequency at each iteration. Finally, we put this edge label into a list so that we can ignore it when we analyze the edge label frequency again.

2) Frequent Pattern Generator: The frequent pattern generator has the following procedure: For each partition, we run `SUBDUE` to discover all the frequent subgraphs having at least two edges from the graph. During the process `SUBDUE` generates a *subgraph file* where it stores the frequent subgraphs it finds. Since a subgraph file is the subgraph frequency list, we gather subgraph files from each partition and generate a graph list. By using a graph mining tool like `GASTON`[13] which solves the set of graphs problem, we find the most frequent subgraph among the graph list.

3) Pattern Replacer: In Fig.7, we show the decompression problem that arises with `SUBDUE`. When the graph in the left is input to `SUBDUE`, it finds a subgraph pattern `x` from the graph. However, the `'-compress'` option of `SUBDUE` produces a graph which is just a connection of two `x` nodes.

This new graph cannot be decompressed because the subgraph `x` has three nodes `a`, `b`, and `c`, and when we say there is a connection between two `x` nodes, it is not clear which inner-nodes of the subgraph `x` take part in the connection. In order to decompress the graph, we need to specify which inner-nodes of a subgraph are connected to exterior-nodes of the subgraph. As we illustrate in Fig.7, we use edge labels for this purpose.

If the new graph produced by `SUBDUE` has an edge whose end node corresponds to a subgraph, then we update the edge label so that it directs which inner-node of the subgraph takes part in the connection. This procedure can be done by comparing the new graph with the original graph and the subgraph description. We modify the `SUBDUE`

[1]The initial edge label is a function of its label of its end nodes.

process so that it uses new edge labels for the purpose of decompression, and call the result the MODIFIED-SUBDUE process.

C. Graph Representation

There are various ways to represent a graph, but among them, we choose to represent a graph using its adjacency matrix because it gave the most compact file size above all when combined with a PAQ compression algorithm[16].

We generate an upper triangular adjacency matrix A as follows: For $(i, j) \in E$, let $l(i, j)$ be the edge label of edge (i, j), $l(i, j) = \phi$ if neither node i nor node j is a subgraph, and $A_{i,j}$ is the (i, j)th element of adjacency matrix A[2].

- If $i < j$ and
  - $l(i, j) = \phi$, then set $A_{i,j} = 1$.
  - $l(i, j) \neq \phi$, then set $A_{i,j} = 3$.
- If $i > j$ and
  - $l(i, j) = \phi$, then set $A_{j,i} = 2$.
  - $l(i, j) \neq \phi$, then set $A_{j,i} = 4$.

In addition to the adjacency matrix, we need the list of node labels and edge labels to represent the graph. We, write the nonzero upper-triangular part of the adjacency matrix in its diagonal order. The, we write the node label in the node order, and the edge label in the order of its occurrence on the adjacency matrix stream. After writing the graph representation in a text file, we compress it with PAQ.

IV. Experimental Results

We tested four algorithms to compress the netlist files. M-SUB is the MODIFIED-SUBDUE process which is a modification of SUBDUE so that the compressed graph could be decompressed. CEDIF is the algorithm we described in section III-B. XML-PPM converts the EDIF netlist into an XML format and compress it using XML-PPM[2], and PAQ compresses the EDIF netlist using paq8o8[16].

Table I shows the compression ratio and the runtime of each algorithm. The compression ratio is defined as

$$\frac{\texttt{EDIF file size-Compressed file size}}{\texttt{EDIF file size}}.$$

The first two columns show the number of nodes and edges in the converted netlist graph. As we can see text compressors, XML-PPM and PAQ, runs much faster compared to graph compressors, M-SUB and CEDIF. But, since they are lossless with respect to the original byte stream, they have worse compression ratio than the graph compressors. Comparing the result of M-SUB and CEDIF, we see that CEDIF runs much faster[3] than M-SUB while they have the similar compression ratio. Moreover, we can see that the compressed result between the first three circuits are similar, meaning that we did not hurt the frequent subgraph patterns by partitioning the graphs.

Finally, the last three columns of Table I shows the stepwise result of CEDIF. Step 1 shows the result after the netlist conversion step in III-A, step 2 shows the result after the

[2]Note that there cannot be a self loop because the netlist conversion does not allow self loop, and the subgraph pattern SUBDUE discovers are induced subgraph of the input graph, and hence, A has 0-diagonals.

[3]M-SUB even failed to give results within a reasonable amount of time which CEDIF did. N/A in Table I means it did not give result within an hour.

Name	$\|N\|$	$\|E\|$	M-SUB	CEDIF	XML-PPM	PAQ	step 1	step 2	step 3
s27	47	55	98.26 (0.89)		91.99 (1)	94.77 (1.89)			
s208	330	411	98.79 (423.78)		94.73 (1)	97.05 (6.14)			
s298	430	578	98.68 (417.69)		94.89 (2)	97.31 (7.58)			
s344	511	639	N/A		95.35 (4)	97.73 (10.28)			
s349	516	647	N/A		95.37 (6)	97.81 (9.30)			
s382	559	744	N/A		95.20 (10)	97.38 (9.92)			
s386	546	737	N/A		95.02 (10)	97.42 (8.67)			
s400	577	772	N/A		95.23 (10)	97.52 (10.11)			
s420	686	863	N/A		95.36 (8)	97.73 (13.28)			
s444	628	836	N/A		95.22 (10)	97.61 (9.89)			
s499	604	832	N/A		95.11 (10)	97.66 (9.05)			
s510	687	891	N/A		95.07 (10)	97.29 (11.23)			
s526	733	1022	N/A		95.25 (10)	97.63 (10.53)			
s635	895	1145	N/A		95.58 (10)	98.16 (14.44)			
s641	1055	1216	N/A		95.47 (10)	97.97 (16.69)			
s713	1120	1319	N/A		95.47 (20)	97.97 (17.41)			
s820	1105	1563	N/A		95.36 (10)	97.57 (14.41)			
s832	1115	1587	N/A		95.39 (20)	97.77 (14.31)			
s838	1398	1767	N/A		95.68 (20)	98.24 (21.39)			
s938	1398	1767	N/A		95.64 (20)	97.94 (20.67)			
s953	1295	1683	N/A		95.51 (20)	97.71 (18.83)			
s967	1314	1723	N/A		95.44 (20)	97.63 (19.41)			
s991	1497	1767	N/A		95.74 (30)	98.34 (23.25)			
s1196	1640	2140	N/A		95.66 (30)	97.73 (22.98)			
s1238	1651	2204	N/A		95.67 (30)	97.94 (23.28)			
s1269	1794	2326	N/A		95.82 (30)	98.20 (26.56)			
s1423	2141	2777	N/A		95.92 (50)	98.42 (30.33)			
s1488	2093	2829	N/A		95.59 (40)	97.84 (30.13)			
s1494	2093	2841	N/A		95.66 (30)	98.18 (31.16)			
s1512	2324	2891	N/A		95.79 (40)	98.14 (36.83)			
s3271	4775	6104	N/A		95.85 (90)	98.24 (73.17)			
s3330	5248	6497	N/A		95.63 (110)	97.95 (81.13)			
s3384	5243	6634	N/A		95.91 (110)	98.55 (76.80)			
s4863	6917	8824	N/A		95.90 (140)	98.41 (95.91)			
s5378	7793	9547	N/A		96.05 (170)	98.35 (123.25)			
s6669	9519	12175	N/A		95.84 (190)	98.60 (146.41)			
s9234	14523	17332	N/A		96.09 (320)	98.78 (218.80)			
s13207	21946	26465	N/A		96.11 (460)	98.83 (327.28)			
s15850	25908	30959	N/A		96.14 (570)	98.82 (389.17)			
s35932	51603	67226	N/A		96.58 (1060)	99.41 (729.47)			
s38417	60887	73978	N/A		96.16 (1360)	98.92 (911.61)			
s38584	58109	74502	N/A		96.01 (1230)	98.65 (825.50)			

TABLE I
COMPRESSION RATIO (%) (RUNTIME (SEC))

graph compression step in III-B, and the step 3 shows the result after graph representation step in III-C.

V. CONCLUSIONS AND FUTURE RESEARCH

We introduced an EDIF netlist compression algorithm based on a graph mining technique which is lossy with respect to the original byte stream but lossless in terms of the circuit structure it contains. We developed a graph partitioning algorithm which attempts to minimize the effect on the frequent subgraph patterns from the original graph in order to achieve scalability on graph mining procedure. We also introduced a way to modify

the graph mining tools so that they can be used for lossless graph compression.

Our compression result is based on a graph mining tool SUBDUE which uses heuristic algorithms to find the frequent subgraph patterns from a single graph. However, since this tool fails to fine all the frequent subgraph patterns in a reasonable amount of time, there is a possibility of improving both the compression ratio and runtime by improving this graph mining procedure. We also see the potential for better representations of compressed graphs, and we leave this as future research.

REFERENCES

[1] J. Adiego, G. Navarro, and P. de la Fuente, "Lempel-Ziv compression of structured text", In Proceedings of the 2004 IEEE Data Compression Conference (DCC 2004), pp.112-121, 2004.

[2] J. Cheney, "Compressing XML with multiplexed hierarchical PPM models", In Proceedings of the 2001 IEEE Data Compression Conference (DCC01), pp.163-172, 2001.

[3] T.M. Cover and J.A. Thomas, *Elements of Information Theory, Second Edition*, Wiley-Interscience, Hoboken, New Jersey, 2006.

[4] P. Grunwald, M.A. Pitt and I.J. Myung, *Advances in Minimum Description Length: Theory and Applications*, MIT Press, Cambridge, Massachusettes, 2005.

[5] ISCAS'89 Benchmark Suite (in EDIF file format), downloadable from http://www.fm.vslib.cz/∼kes/asic/iscas.

[6] W. Li, "XComp: An XML compression tool", Master's thesis, University of Waterloo, 2003.

[7] H. Liefke and D. Suciu, "XMill: an efficient compressor for XML data", In Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, pp.153-164, 2000.

[8] P. Stanford and P. Mancuso, *EDIF: Electronic Design Interchange Format Version 2 0 0*, Electronic Industries Association, ANSI/EIA-548-1988, 1988.

[9] W. Sun, A. Mukherjee, N. Zhang, "A dictionary-based multi-corpora text compression system", In Proceedings of the 2003 IEEE Data Compression Conference (DCC03), p.448, 2003.

[10] D. J. Cook and L. B. Holder, "Substructure Discovery Using Minimum Description Length and Background Knowledge", In Journal of Artificial Intelligence Research, Vol. 1, p.231-255, 1994.

[11] X. Yan and J. Han, "gSpan: Graph-Based Substructure Pattern Mining", Proceedings of the 2002 IEEE International Conference on Data Mining (ICDM02), p.721

[12] J. Huan, W. Wang, and J. Prins, "Efficient Mining of Frequent Subgraphs in the Presence of Isomorphism", Third IEEE International Conference on Data Mining (ICDM03), p.549, 2003.

[13] S. Nijssen and J.N. Kok, "A Quickstart in Frequent Structure Mining can make a Difference", Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining, p.647-652, 2004.

[14] H. Gabow, "Implementation of Algorithms for Maximum Matching on Nonbipartite Graphs", Ph.D. thesis, Stanford University, 1973.

[15] L. Peshkin, "Structure induction by lossless graph compression", Proceedings of the 2007 Data Compression Conference (DCC07), p.53-62, 2007.

[16] Matt Mahoney, "Adaptive Weighing of Context Models for Lossless Data Compression", Florida Tech. Technical Report CS-2005-16, 2005.

[17] N. Deo, B. Litow, "A structural approach to graph compression", in the Proceedings of MFCS Workshop on Communication, pp. 91-101, 1998.

[18] G. Karypis and V. Kumar, "A fast and high quality multilevel scheme for partitioning irregular graphs", SIAM Journal on Scientific Computing, Vol. 20(1), p.359-392, 1998.

[19] M. Kuramochi and G. Karypis, "Finding Frequent Patterns in a Large Sparse Graph", Data Mining and Knowledge Discovery, Vol 11(3), pp. 243-271, 2005.

[20] D.J. Cook and L.B. Holder, *Mining Graph Data*, Wiley-Interscience, Hoboken, New Jersey, 2006.

[21] A. Itai and M. Rodeh, "Representation of Graphs", Acta Informatica, Vol 17, pp.215-219, 1982.

[22] X. He, M-Y. Kao, and H-I. Lu, "Linear-Time Succinct Encodings of Planar Graphs via Canonical Orderings", SIAM Journal on Discrete Mathematics, Vol. 12(3), pp.317-325, 1999.

[23] M. Talamo and P. Vocca, "Representing graphs implicitly using almost optimal space", Discrete Applied Mathematics", Vol. 108(1), pp. 193-210, 2001.