Adaptive Compression of Graph Structured Text

John Gilbert and David M Abrahamson
Department of Computer Science
Trinity College Dublin

{gilberj, david.abrahamson}@cs.tcd.ie

Abstract

In this paper we introduce an adaptive technique for compressing small quantities of text which are organized as a rooted directed graph. We impose a constraint on the technique such that data encountered during a traversal of any valid path through the graph must be recoverable without requiring the expansion of data that is not on the path in question. The technique we present determines the set of nodes y which are guaranteed to be encountered before reaching node x while traversing any valid path in the graph, and uses them as a basis for conditioning an LZW dictionary for the compression/expansion of the data in x. Initial results show that our improved LZW technique reduces the compressed text size by approximately 20% more than regular LZW, and requires only minor modifications to the standard LZW decompression routine.

1 Introduction

Consider an on-line tour guide application for a personal digital assistant where users position themselves at a designated starting point and log onto a website to begin a tour from that location. The pages delivered to a PDA during such tours are fairly small; generally containing a couple of paragraphs of text at most. Each page describes a location/item of current interest along with links to other nearby attractions offering users some choice in tailoring their tour. The collection of pages and links form a directed graph, similar to pages on the world-wide-web, however they remain static once authored and they have a fixed entry point (Figure 1 (i)).

The route of attractions a user visits during their tour can be visualized as a path through the graph. Any route which leads to *The Quays* in our example must start at the *Tourist Office* and directly proceed to *Trinity College Dublin* optionally visiting *The Book of Kells* en-route to *O'Connell Bridge* before reaching the destination. The Tourist Office is guaranteed to be on any path reaching Trinity College, which itself is guaranteed to be on any path reaching either the Book of Kells or O'Connell Bridge. Finally, O'Connell Bridge is guaranteed to be on any path reaching The Quays. This relation, where one node in a graph is guaranteed to be present on any path reaching another is called *dominance*, and its transitive reduction gives rise to a tree structure called an *immediate dominator tree* (Figure 1 (ii)). While compression can be applied to the text for each page individually using well known techniques [2], we propose

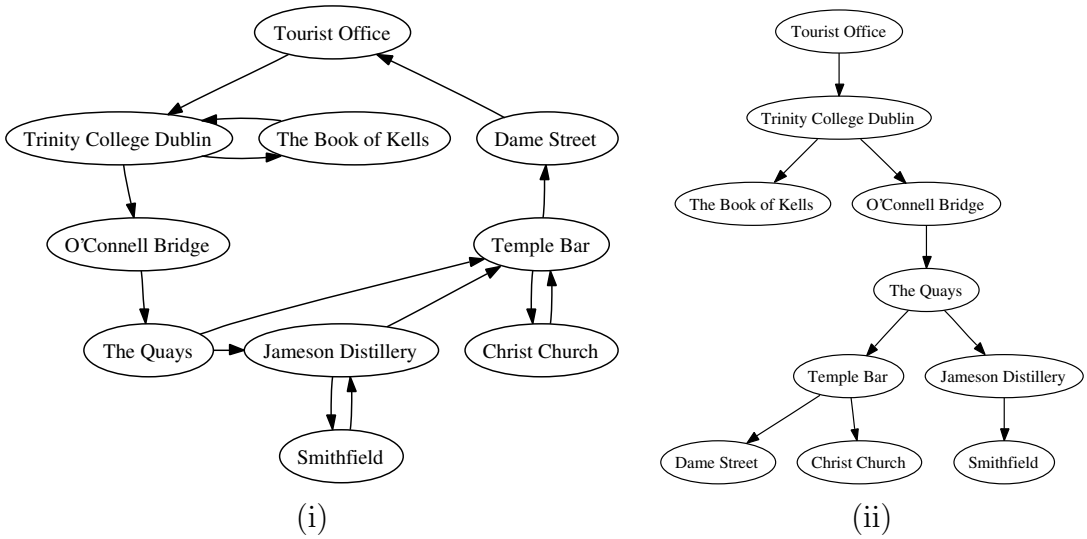


Figure 1: Example of graph structured text (left) and the associated immediate dominator tree (right)

exploiting the inter-page context identified by dominance to improve results when using adaptive dictionary based compression methods.

The remainder of this paper is structured as follows. First we briefly review the LZW algorithm and the structure of an LZW dictionary during decompression. Next we give formal definitions of a rooted directed graph, the dominance and immediate dominance relations, and we discuss the relationship between an immediate dominator tree and the graph from which it was derived. With the appropriate background material covered, section 4 describes our new approach to compression in detail. Results and related work are presented in sections 5 and 6 respectively, before our conclusions are made in the final section.

2 LZW

LZW [11] is a popular text-compression algorithm. It is classified as lossless, dictionary-based and adaptive. A block of LZW-coded data is comprised of a sequence of fixed-length n-bit indices into a dictionary of 2^n phrases, constructed in the following way: Initially the dictionary contains only the individual symbols from the input alphabet $\sum$. At each step the dictionary is searched to find the longest matching phrase which is a prefix of the input data. The index for this phrase is output as the compressed encoding, and a new entry, consisting of the next symbol from the input concatenated to the end of the phrase just matched, is inserted into the dictionary. Coding continues in this fashion, restarting each time from the last unmatched symbol in the input. When the dictionary is full no more entries can be added and the remaining input data is coded using the dynamically-constructed dictionary.

Expansion of LZW coded data starts with the initial dictionary used for compression. Each codeword in the input is used as an index to the dictionary and its

Index	Parent	Suffix	Phrase
0	-1	α	α
1	-1	β	β
2	0	α	$\alpha\alpha$
3	0	β	$\alpha\beta$
4	1	α	$\beta\alpha$
5	2	β	$\alpha\alpha\beta$
6	-	-	-
7	-	-	-

Figure 2: LZW dictionary after compression/expansion of the string $\alpha\alpha\beta\alpha\alpha\beta$

corresponding phrase is output to the decompressed stream. Then the first symbol from the phrase in the dictionary indexed by the next codeword in the input stream is appended to the phrase just decompressed, and this sequence is inserted as a new entry in the dictionary. In this way, the decompressor maintains the same dictionary as that generated during compression. The dynamically constructed LZW dictionary encodes a history of previously encountered phrases in the input stream and gives rise to compression when single codes (dictionary indices) are output in place of multiple symbols from the input stream during coding.

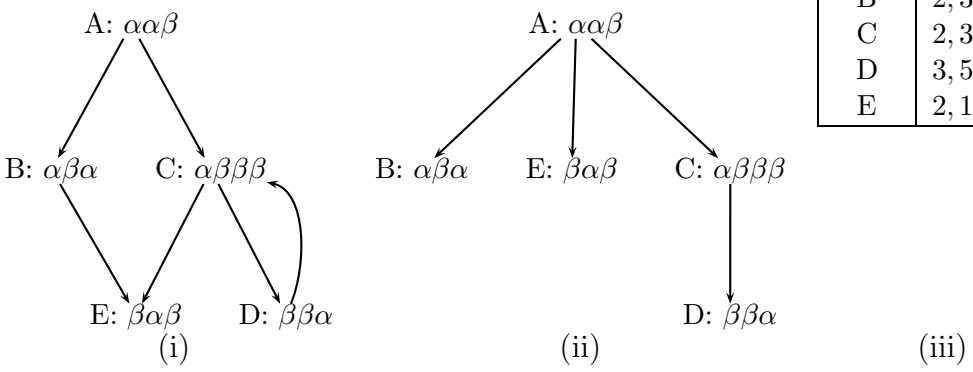
As an example assume $\sum = \{\alpha, \beta\}$ and $n = 3$. Then, after recovering the compressed version of the string $\alpha\alpha\beta\alpha\alpha\beta$ (which consists of the sequence of LZW indices 0,0,1,2,1), the LZW dictionary will appear as shown in Figure 2. Our dictionary representation consists of two arrays, one of pointers back into the dictionary (`Parent`), and one of symbols taken from $\sum$ (`Suffix`). The first section is the initial dictionary, containing entries for each symbol in $\sum$. The second section contains the adaptations which are entered as part of the standard LZW encoding/decoding procedure. The final section consists of unused dictionary space. A phrase in the dictionary located at index w consists of the phrase in the dictionary located at `Parent[`w`]`, concatenated with `Suffix[`w`]`[1].

3 Immediate Dominance

Let $G = (V, E, root)$ be a directed rooted graph where V is the set of vertices/nodes, $E \subseteq V \times V$ is the set of edges, and $root \in V$ is a distinguished node. Given $x, y \in V$ we say x dominates y (x dom y) if every path from $root$ to y in G passes through x. Every node dominates itself and every node is dominated by $root$. Node x *strictly* dominates y (x sdom y) if x dom y and $x \neq y$. Node x immediately dominates y (x idom y) if x sdom y and x does not dominate any other dominator of y.

Every node other than $root$ has a unique immediate dominator. This relation may be represented by a dominator tree stemming from $root$ with edges representing the idom relation between nodes. A fast algorithm for computing immediate dominators in a directed rooted graph is available [6]. Intuitively, the immediate dominator of y is the most recent node x which, regardless of the path taken, is guaranteed to have been encountered before reaching y while traversing G from $root$. Figure 3

[1]The phrase represented at index '-1' in the dictionary is ϵ, the empty string.



Node	Encoding
A	1, 0, 0, 1
B	2, 3, 0
C	2, 3, 1, 1
D	3, 5, 0
E	2, 1, 3

Figure 3: A rooted directed graph, its associated immediate dominator tree, and the Graph LZW encoded data for each node

shows an example graph of five nodes labeled A to E (i) and its associated immediate dominator tree (ii).

Arrival at a node p while traversing G starting from *root* occurs in the following way with respect to the immediate dominator tree: (i) it arrives directly from p's immediate dominator; or (ii) it arrives directly from some descendent of p's immediate dominator. The justification for this is as follows: a traversal may not pass directly from a node at level i in the tree to node p at level $i + 2$, as doing so would imply that p's immediate dominator on level $i+1$ did not dominate node p. Furthermore, a traversal may only pass across a tree from node q to node p if their common ancestor in the tree is p's immediate dominator (otherwise it would again bypass traversing the immediate dominator of p). It is easy to verify that traversal may pass from a node to one of its ancestors in the tree due to the existence of back edges in G.

4 Graph LZW

The key insight here is that the nodes which dominate a given node x constitute the set from which we can obtain textual-context for the adaptive compression of the contents of x; furthermore, the structure imposed on these dominators by the immediate dominator tree indicates the order in which they will have been encountered on all paths leading to x. However, since the dictionary used during encoding/decoding is conditioned by the contents of only those nodes guaranteed to have been encountered previously on all paths reaching the node in question, decoding cannot start at any arbitrary position in the graph.

Before encoding the contents of the root node in the immediate dominator tree the dictionary is pre-initialized to contain entries for the alphabet $\sum$. The tree is then traversed in pre-order and the contents of each node are compressed using the dictionary which resulted from coding its parent. The deeper a node is in the tree, the

Index	end_of_adapt
0	2
1	-
.	.
.	.
k	-

Index	Parent	Suffix	Phrase
0	-1	α	α
1	-1	β	β
2	-	-	-
.	.	.	.
2^3-1	-	-	-

Figure 4: Initial structure of dictionary during Graph LZW expansion

Index	end_of_adapt
0	2
1	4
2	6
3	7
4	-
.	.
.	.
k	-

Index	Parent	Suffix	Phrase
0	-1	α	α
1	-1	β	β
2	0	α	$\alpha\alpha$
3	0	β	$\alpha\beta$
4	3	β	$\alpha\beta\beta$
5	1	β	$\beta\beta$
6	5	α	$\beta\beta\alpha$
7	-	-	-

Figure 5: Dictionary structure after traversing nodes A, C and D

more context we have available for compression. But care is needed—for each node x, before decoding starts we must ensure the dictionary contains only those entries added up to the point where x's immediate dominator made its final adaptation.

In the previous section we established that flow passes through a graph from node to node such that we move down the corresponding immediate dominator tree one level at a time, move from the current node to one of its ancestors' children or move from the current node to one of its ancestors. Hence the LZW dictionary may be maintained appropriately at decode time as follows.

We introduce an additional array, `end_of_adapt`, with capacity for storing k n-bit entries where k is the maximum immediate dominator tree depth to be supported. `end_of_adapt[0]` is initialized to $|\sum|$. The next available dictionary index, after node x at depth i in the immediate dominator tree has been decoded, is recorded in `end_of_adapt[`i`]`. Before the next node y at depth j in the dominator tree is decoded, all entries in the dictionary from `end_of_adapt[`$j-1$`]` onwards are removed and then the node y is decoded, adapting the dictionary while the compressed data is expanded and finally setting `end_of_adapt[j]` to the next available dictionary index. Thus we need to store the depth of each node in the immediate dominator tree along with its encoded data. Since dictionary maintenance must occur prior to decoding each node, this is stored as the first index in the encoded block.

As an example, the compressed encoding for each node in the graph introduced in Figure 3 (i), with its associated immediate dominator tree (ii), is shown on the right (iii). Note: the first value in the encoding is the $\lceil log_2 k \rceil$-bit depth of the node in the immediate dominator tree; while the remaining values are n-bit LZW dictionary indices for the compressed data. Consider traversing the path $A \rightarrow C \rightarrow D \rightarrow C \rightarrow ....$ We begin with the initial dictionary shown in Figure 4. After visiting nodes A, C and D the dictionary will be as presented in Figure 5, where each successive section contains the adaptations contributed by the nodes A, C and D

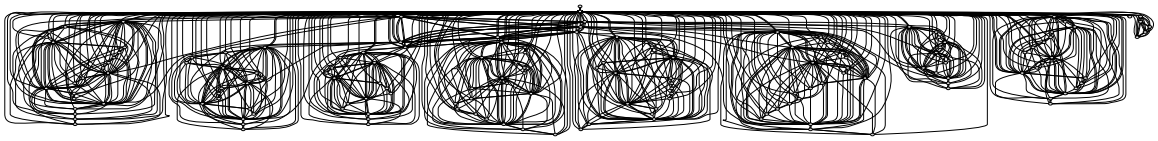


Figure 6: Connectivity graph extracted from our snapshot of wap.sciam.com (The *Scientific American* website, mobile edition)

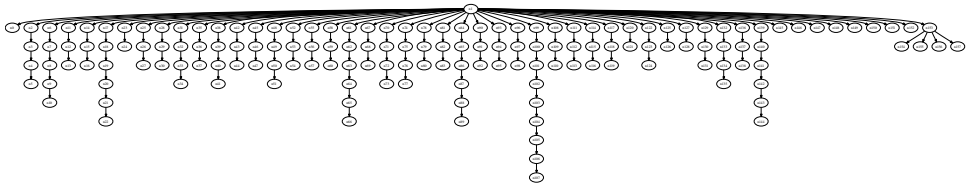


Figure 7: Immediate dominator tree derived from the graph shown in Figure 6

respectively. As we traverse the edge $D \rightarrow C$ we must remove the dictionary entries previously contributed by nodes C and D (trivially implemented by resetting the pointer identifying the next available slot in the dictionary). This happens prior to revisiting C thus ensuring that only those entries guaranteed to have been added before arriving at C remain in the dictionary. That is, all entries from `end_of_adapt[1]` onwards are removed from the dictionary (which is where the adaptations based on the contents of A finish), C is expanded and finally `end_of_adapt[2]` is set to the next available index in the dictionary—where the adaptations based on the contents of C finish—which is 6.

5 Results

We applied our technique to a collection of PDA websites which were downloaded in July and September 2007 using the *wget* program (generally invoked as *wget -r -l 1000 http://homepage-address*). The full list of websites we used is given in Figure 8, where summary information characterizing the websites' respective file sizes is also included. Once downloaded we extracted the connectivity graph for each website using a collection of perl scripts to identify the edges exiting each file and targeting another file within that website. This is the input that was given to our prototype compression implementation. In Figures 6 and 7 we show the connectivity graph and the associated immediate dominator tree for one of the websites.

We applied regular LZW to each of the files individually, and Graph LZW to the entire set of files for each website, the results of which are given in Figure 9. While regular LZW gives its best compression when using 10 and 11-bit dictionary indices, our Graph LZW technique doesn't run out of context to condition the models used for compression until it employs substantially larger dictionaries requiring 12 and 13-bit indices, demonstrating clearly that exploiting inter-page context is of great benefit to compressing graph-structured text. The summary figures account for all overheads in the compressed data encodings.

PDA Website URI	#Files	Corpus Size (kB)	Average File Size(kB)
www.australianit.news.com.au/wireless	59	275.38	4.67
www.news.com.au/wireless	78	231.81	2.97
wap.sciam.com	158	528.78	3.35
news.bbc.co.uk/low/english/pda	151	882.18	5.84
news.bbc.co.uk/low/english/pda_sport	182	1098.16	6.03
www.rte.ie/pda/entertainment	4150	17376.19	4.19

Figure 8: PDA website benchmark data

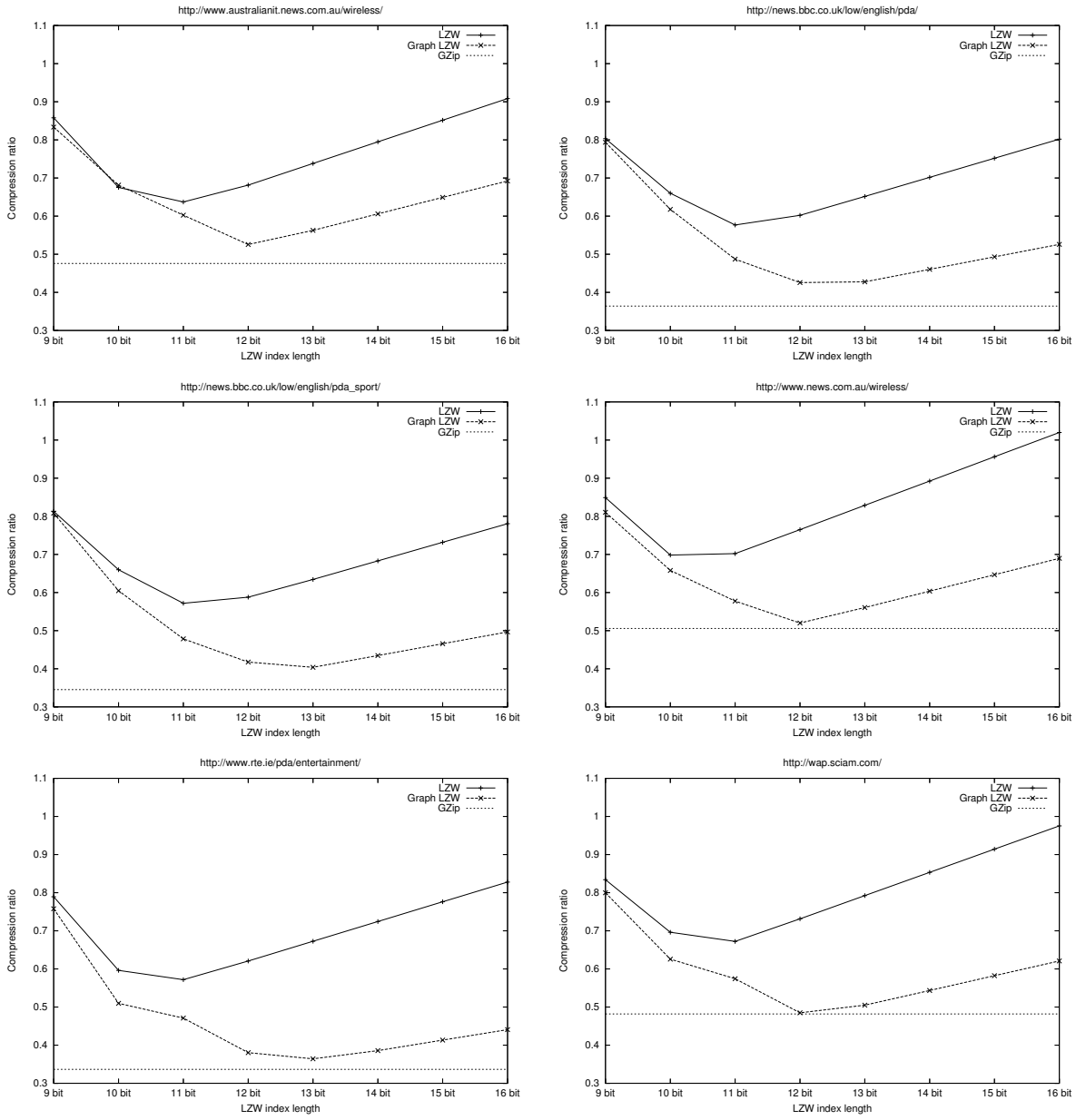


Figure 9: Compression ratios (compressed size/uncompressed size) achieved at various dictionary sizes for both regular LZW and Graph LZW applied to six PDA websites

gzip [8], a popular compression program, employs a variant of LZ77 combined with Huffman encoding to attain excellent compression results. It is used in the widely deployed *mod_gzip* [1] extension available for the Apache webserver which compresses data before sending it via HTTP to clients capable of expanding data prior to rendering. We have included the average compression ratio achieved by *gzip* on our data in Figure 9. Whereas *gzip* is a carefully engineered compression program, our Graph LZW and the results given here are simply the kernel of an idea. Nevertheless we achieve competitive compression results and the expansion routine for Graph LZW is substantially less involved than that of *gzip*. In our prototype implementation the decoder required the following simple alterations to the regular LZW decoding algorithm

- introduction of an additional array, `end_of_adapt`
- one read from `end_of_adapt` before expansion of a node to identify the position where adaptations should occur in the dictionary
- one write to `end_of_adapt` after expansion of a node has completed to update the contents of the array

Inspired by *gzip*'s combination of an adaptive dictionary technique with a semi-adaptive code allocation phase, we investigated using a canonical Huffman code to represent the dictionary indices used by our Graph LZW algorithm. Using this approach we found that although the compression ratios improved over our fixed-length allocation of dictionary indices, the compression ratios did not beat *gzip* on average. Careful engineering and an appropriate combination of our technique with a better form of data encoding might yield compression ratios which improve on those achieved by *gzip*, however we believe the overhead required in the decoder would not represent a justifiable penalty for the marginal improvements likely to arise.

6 Related Work

To the best of our knowledge we are the first to propose compressing the content of nodes in a graph by employing an adaptive model which has been conditioned using inter-node context. Our previous work presented an LZW-based technique targeting a computer program's object code which exploited the dominance relation to identify inter-cache line context for compressing instruction cache lines [5]. In that paper we derived a data structure (the *compulsory miss tree*) for representing a partial ordering in which compulsory misses in an instruction cache would occur, and used it to condition the LZW dictionary for compression/expansion of each cache line in the program's code. While the underlying source of context based on dominance was the same as that presented in this paper, the method by which it was exploited to condition and maintain the LZW dictionary during the compression/expansion of a program was dramatically different to that presented here: space in the LZW dictionary was statically allocated for each cache line contributing context to its

children in the compulsory miss tree, whereas the management of the dictionary in this paper treats the dictionary more like a stack.

All previous work we encountered in the literature which related to compression involving graphs focused on efficiently storing their structure in a more compact manner than that offered by a straightforward representation using adjacency matrices or adjacency lists. Some of the techniques targeted web graphs—which represent the link structure of the world-wide-web [3, 7, 9], while others targeted trees represented by a stationary ergodic source [4]. In some cases the authors stored the URI of a node as content along with the node and also required this data to be represented in a space efficient way [10]. Typically, to allow direct and efficient access into the graph, the standard graph data structure operations must be implemented over the compressed encoding. This is in stark contrast to our approach which only requires that valid paths be traversable through the graph. A more restricted view is that presented in [4] where LZW is used to compress trees, but expansion of the trees must occur in their entirety due to the breadth-first search parsing of entries in the LZW dictionary. Like our approach, all methods we have seen operating on trees and graphs assume the structure is static and any modification requires a re-computation of the compressed representation.

7 Conclusion

Ziv-Lempel methods produce optimal coding as the size of the input tends to infinity. Our technique increases the effective length of sequences being compressed within a set of textual nodes organized as a directed rooted graph structure, giving a dramatic increase in the compression ratios attained. We do this by exploiting the structural properties of directed graphs which often provide a good abstraction over data encountered in computer systems, such as pages in the world-wide-web or control flow graphs encountered by an optimizing compiler.

When compared with the LZW technique our Graph LZW algorithm trades additional analysis undertaken at compression time for improved compression ratios. Despite our superior compression, the modifications to a standard LZW decompressor to support our approach are relatively minor and do not contribute any significant time overhead to the decoding routine. Notwithstanding the simplicity of our decoding routine, the results we have presented are comparable with those achieved by *gzip* which requires a considerably more complex decoder combining both static Huffman and LZ77 decoding routines.

Acknowledgments

We are grateful to Edsko de Vries for helpful discussions and for critical feedback on early drafts of this paper. The work of John Gilbert was supported by a Trinity College Dublin Postgraduate Award.

References

[1] mod gzip. http://sourceforge.net/projects/mod-gzip/.

[2] Timothy C. Bell, John G. Cleary, and Ian H. Witten. *Text Compression*. Prentice Hall, 1990.

[3] Paolo Boldi and Sebastiano Vigna. The webgraph framework ii: Codes for the world-wide web. In *DCC '04: Proceedings of the Conference on Data Compression*, page 528, Washington, DC, USA, 2004. IEEE Computer Society.

[4] S. Chen and J. H. Reif. Efficient lossless compression of trees and graphs. In *DCC '96: Proceedings of the Conference on Data Compression*, page 428, Washington, DC, USA, 1996. IEEE Computer Society.

[5] John Gilbert and David M. Abrahamson. Adaptive object code compression. In *CASES '06: Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems*, pages 282–292, New York, NY, USA, 2006. ACM Press.

[6] Thomas Lengauer and Robert Endre Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Trans. Program. Lang. Syst.*, 1(1):121–141, 1979.

[7] Fang-Yie Leu and Yao-Chung Fan. Compressing a directed massive graph using small world model. In *DCC '04: Proceedings of the Conference on Data Compression*, page 550, Washington, DC, USA, 2004. IEEE Computer Society.

[8] Jean loup Gailly and Mark Adler. Gnu zip. http://www.gzip.org.

[9] A. Mahdian, H. Khalili, E. Nourbakhsh, and M. Ghodsi. Web graph compression by edge elimination. In *DCC '06: Proceedings of the Data Compression Conference (DCC'06)*, page 459, Washington, DC, USA, 2006. IEEE Computer Society.

[10] Torsten Suel and Jun Yuan. Compressing the graph structure of the web. In *DCC '01: Proceedings of the Data Compression Conference (DCC '01)*, page 213, Washington, DC, USA, 2001. IEEE Computer Society.

[11] Terry A. Welch. A technique for high-performance data compression. *IEEE Computer*, 17(6):8–19, 1984.