Towards Compressing Web Graphs

Towards Compressing Web Graphs

Micah Adler* Michael Mitzenmacher†

Abstract

In this paper, we consider the problem of compressing graphs of the link structure of the World Wide Web. We provide efficient algorithms for such compression that are motivated by recently proposed random graph models for describing the Web. The algorithms are based on reducing the compression problem to the problem of finding a minimum spanning tree in a directed graph related to the original link graph. The performance of the algorithms on graphs generated by the random graph models suggests that by taking advantage of the link structure of the Web, one may achieve significantly better compression than natural Huffman-based schemes. We also provide hardness results demonstrating limitations on natural extensions of our approach.

*University of Massachusetts, Department of Computer Science, University of Massachusetts, Amherst, MA 01003-4610. E-mail: micah@cs.umass.edu.

†Harvard University, Division of Engineering and Applied Sciences, 33 Oxford St., Cambridge, MA 02138. Supported in part by a research grant from the Alfred P. Sloan Foundation, NSF grant CDA-94-01024, and an equipment grant from Compaq Computer Corporation. E-mail: michaelm@eecs.harvard.edu.

1 Introduction

A snapshot of the World Wide Web can be thought of as a graph, with Web pages represented by nodes and hyperlinks represented by directed edges. This representation has been used for a wide variety of Web algorithms, including algorithms for ranking pages based on their connectivity [11, 3] and finding natural communities of pages on a shared topic [13]. Indeed, at least one major search engine has designed a tool called the connectivity server for storing the Web graph [2, 4].

Given this previous work, a natural question to ask is how well the Web graph and Web-like graphs can be compressed, in order to save on the memory required to store or transfer such graphs. Good compression requires using the structural properties of the Web graph, and hence an important first step is understanding this structure. Previous work gives us important insights. It is clear that the Web graph appears to be significantly different from the likely graphs resulting from traditional random graph models. In particular, there appear to be natural clusters of related pages with similar connections. Hence, in [12, 14], a new random graph model was introduced with these clustering properties. The basis of this model is that pages and links enter and leave the system dynamically, and new pages may link to other pages by finding one or more *reference* pages and copying links from these references.

Recent studies of the Web graph suggest that the structure of the Web is actually more complex than this random graph model; see, for example, a study based on a recent snapshot of the Web from Altavista [4]. However, as a first approximation, this model captures important high level behavior, and it may be especially suitable for the large components of the Web graph, or for specific subdomains, such as all the pages within a given university. Hence, in this paper we focus on variations of this graph model; experiments on complete Web data will be covered in future work. Since achieving good compression is strongly related to finding structure, we expect our compression work eventually to yield further insights into the structure of the Web graphs.

Our primary results are the following:

- We provide a compression algorithm based on finding pages with many shared links. This dependence on shared links makes our algorithm particularly well suited to graph models that employ copying links from reference pages, but we also expect that our algorithm will work well with any graph structure that involves significant similarities in shared links. The algorithm requires solving a directed minimum spanning tree on a graph associated with the original graph. Under appropriate assumptions concerning the degree distribution of the Web graph to be compressed, the running time of our algorithm is $O(n \log n)$, where n is the number of nodes in the Web graph.

- We provide hardness results demonstrating that several natural extensions of our algorithm are NP-Hard.

- We demonstrate the effectiveness of our approach on a testbed of random graphs derived from the random graph models that motivate our approach. Our results appear significantly better than natural Huffman-based alternatives.

1.1 Framework

When we discuss compressing Web-like graphs, there are actually a variety of distinct situations we may wish to consider:

1. Compressing the underlying graph for storage or transmission, up to isomorphism. This setting would be useful if we want to store just the graph structure itself.

2. Compressing the underlying graph for storage or transmission, maintaining a given ordering of the nodes. As an example of this setting, we might order the nodes according to the sorted order of the URLs (so that the URLs can be compressed by delta encoding, as in [2]).

3. Compressing the underlying graph for use in its compressed form. That is, we desire a compressed form of the graph that still allows for efficient computation on the compressed form.

Our primary focus in the paper is the second setting, where we are given a node ordering and are concerned solely with overall compression. However, the three problems are clearly related, and we will suggest connections between the variations as they arise.

1.2 A Web graph model

We reiterate that in this paper, rather than compress actual subgraphs of the Web, our focus is a recently proposed Web graph model that captures certain aspects of Web graphs. We have thus far only tested the algorithm on a single subgraph of the real Web graph (on which, as we show, performance is quite good); we expect to test our compression scheme on more extensive real Web data in future work. We also believe our experiments on the Web graph model we examine are interesting in their own right.

The model, taken primarily from [14], uses the following basic outline. The graph evolves over time by associated node and edge creation and deletion processes. The intuition suggested from [14] is the following: "*A new page adds links by picking an existing page, and copying some links from that page to itself.*" For example, a new page v might examine the outedges from a page w and link to a subset of the pages that w links to; we call this *copying outedges*. This intuition is based on the idea that a user decides what pages to link a new page to based on a page or pages that the user already likes.

Given this framework, there are a variety of possible variations, depending on the specifics of the edge creation and deletion process as well as the copy process. We specify the model we use here. We begin with an initial graph of n_0 nodes, with each having d_0 outedges connected to nodes chosen uniformly at random. There is no deletion process, only a node creation process. One new node is created each time step to a total of n nodes. The creation process is determined by probability distributions $\mathcal{A}, \mathcal{B}, \mathcal{C}, \mathcal{D}, \mathcal{E}, \mathcal{F}$. The distribution $\mathcal{A}$ provides a number a, such that v is given a outedges with each edge pointing to a node chosen uniformly at random from all nodes existing at that time. Similarly, $\mathcal{B}$ provides a number such that v copies outedges from b nodes, again chosen uniformly at random. The distribution $\mathcal{C}$ yields a probability; for each of the b nodes $w_1, \ldots, w_b$ chosen to copy from, a probability

is independently chosen from $\mathcal{C}$, and each outedge from w_i is copied independently with the probability determined from $\mathcal{C}$.[1] Distributions $\mathcal{D}, \mathcal{E}$, and $\mathcal{F}$ are analogous to $\mathcal{A}, \mathcal{B}$, and $\mathcal{C}$ respectively, except that they determine the inedges for a new page.

Graphs based on this copying procedure do not appear to have been given a general name. The resulting graphs may have more specific structure than power-law graphs, such as those suggested in [1]. We suggest the name *copy graphs* for families of evolving random graphs using copying operations of this type.

2 A baseline Huffman-based scheme

Experiments have demonstrated that the indegrees and outdegrees of Web pages follow a Zipfian distribution [1, 14, 4]. That is, the fraction of pages with indegree j is roughly proportional to $1/j^{\alpha}$ for some fixed constant α, and similarly the fraction of pages with outdegree j is roughly proportional to $1/j^{\beta}$ for some fixed constant β. One of the features of the copy graph model is that it can yield graphs with such Zipfian distributions [14].

Given the large variance in degrees, it is natural to consider Huffman-based compression schemes. A simple such scheme would go through the nodes in order and list the destination of each outedge directed from that node. Each page would be assigned a Huffman codeword based on its indegree. To separate the outedges of each node we could utilize a special stop symbol, which would appear $n-1$ times if there are n pages. An end of file symbol would denote the end of the edge list. In Appendix 1, we give an expression for the number of bits required to represent a graph using this kind of compression, provided that the indegree follows a Zipfian distribution with $\alpha > 2$. The expression demonstrates that this kind of compression requires asymptotically the same number of bits as not using Huffman compression.

Many simple variations are possible. The compression scheme could also be based on the edges directed into each node instead of the edges directed out from each node. Whether inedges or outedges is a better choice depends on which has higher variance. In the case where we only need to store an isomorphism of the graph, we might avoid the stop symbol. Instead we can send an implicit or explicit representation of the outdegree distribution, sort the nodes by outdegree, and list the outedges for each node as before without the stop symbol. Also, we note that an advantage of this compressed representation is that it is extremely simple and convenient; it could naturally be used in the framework of the connectivity server [2], or in any system that wanted efficient computation on the compressed form of the graph.

This approach achieves significant compression with little complexity, and thus we shall use it as a baseline for algorithms that compress the Web graph. Note that this Huffman-based scheme ignores the natural clustering structure induced in copy graphs. We next examine how to take advantage of this structure and show via experiments that this does in fact yield substantially greater compression.

[1] We note that another possibility is to use a distribution $\mathcal{C}$ to determine the number of outedges to copy from each w_i, as done in [14]. We believe that our approach may be easier to analyze in the future, as it avoids the problem of choosing a number of links to copy that is greater than the number of outedges from a page. Our results apply to both possibilities.

3 The Find-Reference algorithm

Our basic algorithm is based on the following insight: given the random graph model, a natural approach to compress a Web graph is to try to reconstruct how the graph developed according to that model. That is, if we knew the history of how the graph was created (according to the random graph model), we might achieve a great deal of compression by working with this history instead of the actual graph. In practice, we attempt to find nodes that share several common outedges, corresponding to cases where one node might have copied the links of another. Once an appropriate neighbor is identified, the difference, or delta, between the outedges of the two nodes can be identified. When node i is compressed in this way using node j, we say that node j is a *reference* for node i.

For example, if node i is labeled as a reference of node j, we can include a 0/1 bit vector denoting which outedges of node j are also outedges of node i. Other outedges of i can then be separately identified, say using $\lceil \log n \rceil$ bits in an n node graph. Of course we must also identify i, which is another $\lceil \log n \rceil$ bits. Let $N(i)$ and $N(j)$ represent the set of outedges for node i and node j respectively. The cost of compressing node i using node j as a reference with this scheme is then

$$\text{cost}(i, j) = \text{out-deg}(j) + \lceil \log n \rceil \cdot (|N(i) - N(j)| + 1) .$$

Given a description of a graph in this kind of compressed format, consider how we would determine where a link from node i encoded using node j as a reference actually points. If the corresponding link from node j is encoded using another node k as a reference, then we would need to determine where the corresponding link from node k points. Eventually, we must reach a link that is encoded without using a reference node. In order to satisfy this requirement, we shall not allow any cycles among references. For example, we shall not allow i to be compressed using j as a reference, j to be compressed using k as a reference, and k to be compressed using i as a reference.

An intermediate structure that Find-Reference uses is the *affinity graph* G_S for the given Web graph G_W. Specifically, the nodes of G_S are the same as the nodes of G_W. We set $w(i, j)$, the weight of the directed edge from node i to node j, to be the cost of compressing node i using node j as a reference. We add to the affinity graph a *root node* r to which every other node has a directed edge and from which there are no directed edges. The weight of the edge from i to r is the cost of compressing i without using any other node as a reference. We assume that node i has a directed edge to node j if and only if $w(i, j) < w(i, r)$.

Given a Web graph, the algorithm Find-Reference first computes the corresponding affinity graph for the given cost function, and then finds an optimal set of references under the restrictions that (a) each node has at most one reference, and (b) there are no cycles among references. The problem of finding the globally best mapping from nodes to references (or to the dummy node) is equivalent to finding the minimum weight directed spanning tree with root r on the affinity graph. Thus, a high level description of the compression algorithm is as follows:

Algorithm Find-Reference

- Given a Web graph G_W, compute the corresponding affinity graph G_S.

- Compute a minimum directed spanning tree D rooted at r for the graph G_S.
- Compress the graph G_W, where node i uses node j as a reference if and only if node i points to node j in D.

Theorem 1 *For a Web graph G_W, let n be the number of nodes in G_W, and let $t_{G_W}(i)$ be the indegree of node i. Algorithm* FIND-REFERENCE *can be realized to run in time $O\left(\sum_{i=1}^{n}(t_{G_W}(i)^2) + n \log n\right)$.*

Proof: The affinity graph G_S can be computed from the original graph G_W by using a matrix multiplication. When G_W is a Web graph, we expect it to be sparse, and so we describe the algorithm in terms of a sparse matrix multiplication. Let M represent the adjacency matrix of G_W. It is easy to verify that $(MM^T)_{ij}$ is the number of nodes that both i and j have outedges to. The matrix (MM^T) can be computed in time $O\left(\sum_{i=1}^{n} t_{G_W}(i)^2\right)$, assuming that we compute a list of the non-zero entries of (MM^T). We also compute an array R, where $R[j] = \text{out-deg}(j)$. This requires time $O(n+m)$, where $m = \sum_{i=1}^{n} t_{G_W}(i)$ is the number of edges in G_W. Given $(MM^T)_{ij}$ and $R[j]$, $\text{cost}(i,j)$ can be computed in constant time.

Note that there will never be an edge from i to j in G_S unless nodes i and j in G_W have an outgoing edge to at least one shared neighbor. Thus, to compute the set of edges in G_S, we only need to compute $\text{cost}(i,r)$ for every i, and then for every edge from i to j such that $(MM^T)_{ij} > 0$, compute $\text{cost}(i,j)$, and compare it to $\text{cost}(i,r)$. The set of edges in G_S is $\{(i,j) \text{ s.t. } \text{cost}(i,j) < \text{cost}(i,r)\}$ and the set of edges from every other vertex to r. This also gives us the weight of each edge in G_S. Since there can be at most $\sum_{i=1}^{n} t_{G_W}(i)^2$ nonzero entries in (MM^T), the total time required to compute the graph G_S is $O\left(n + \sum_{i=1}^{n} t_{G_W}(i)^2\right)$.

Computing a minimum directed spanning tree with root r in a directed graph is generally referred to in the literature as a *branching* with root r.[2] For information on branchings, see for example [6, 8, 10, 16]. Minimum spanning trees in directed graphs with x nodes and y edges can be found deterministically in time $O(x \log x + y)$ [8]. A simpler algorithm that runs in time $O(y \log x)$ is suitable for the case of sparse graphs [16, 6], which will generally be the case in our context. Since the total number of edges in G_S is at most $\sum_{i=1}^{n} t_{G_W}(i)^2 + n$, the total time required to compute the minimum directed spanning tree in G_S is $O\left(n \log n + \sum_{i=1}^{n} t_{G_W}(i)^2\right)$.

All that remains is to perform the compression using the computed directed tree to specify a reference for each node. To do this, we compute for each node i with reference node j a linked list of outedges that i and j have in common. This set of lists can be computed in time $O\left(\sum_{i=1}^{n} t_{G_W}(i)^2\right)$. With the list of edges that i and j have in common, the compressed version of node i can be computed in time $O(\text{out-deg}(i))$. Thus, the entire algorithm runs in time $O\left(n \log n + \sum_{i=1}^{n} t_{G_W}(i)^2\right)$. □

Note that the performance of this algorithm is particularly good when G_W is sparse, as we expect of Web graphs. For example, if the distribution of indegrees in G_W is Zipfian with $\alpha > 3$, then $\sum_{i=1}^{n} t_{G_W}(i)^2 = O(n)$.

[2] Branchings generally refer to the (equivalent) maximum weight problem. They are sometimes also referred to as arborescences.

We point out that we have found a similar idea to the algorithm FIND-REFERENCE alluded to in [5], in the context of compressing tables of data, where one column can be used to compress another. The authors mention that the problem can be reduced to a minimum spanning tree problem (in their case, edges are undirected).

3.1 Additional improvements and related problems

In practice, after we have found the references via the directed minimum spanning tree, there are various improvements that can be implemented. First, we may wish to find additional references for greater compression. This can be done by stripping edges from the original graph handled by the first references, re-calculating the cost function accordingly, and re-running the algorithm. This algorithm is not optimal, however, since we may obtain better compression if we choose the references of the first stage keeping in mind that we have further stages coming. Although it appears that a better approach would be to find multiple references simultaneously instead of in stages, in general finding multiple references in an optimal manner is a hard problem, as we show in Section 4. Hence the stage attack is likely to be the most efficient and effective in practice.

Once we have found the best references, we may again use a Huffman encoding to handle the edges not covered by references. Note that by doing this, we invalidate the cost function we used to determine the references, so that the set of references may not be optimal. However, until we choose the references, we cannot determine the cost of edges not covered by references, so it seems difficult to take this into account properly in the cost function. One possibility is to attempt to approximate this effect in the cost function. Another is to apply a heuristic approach such as hill-climbing to find the best references. Since this process is likely to be time-consuming, starting with a good solution from our algorithm may prove effective. The gain from this consideration is likely to be small, and so in practice it can probably be ignored.

Other possibilities include using different compressed representations. We have suggested using a bit vector to denote which links a node has copied from its reference. These bit vectors can be Huffman encoded; alternatively, a run-length encoding might be applicable here. Also, if a node only copies a small fraction of the links from its reference, a list of the copied links may be more efficient than a bit vector. As usual with compression schemes, there are a variety of possible enhancements that may slightly improve compression. However, we believe the main concept of using similar pages for compression provides the bulk of the benefit.

While our algorithm is described for the problem of storing Web graphs, we believe these techniques can also be useful when we wish to compute or use the compressed form. The main potential problem is that in order to find the inedges or outedges of a node, one may have to go through multiple references in the directed minimum spanning tree, which may take more time than is desired for fast computation. To bound the number of references to pass through in our single-reference setting it is sufficient to bound the depth of the directed minimum spanning tree we find on the affinity graph. Unfortunately, finding the optimal directed minimum spanning tree of bounded depth is NP-hard; for example, if we allow depth at most two, then the problem of finding the optimal directed minimum spanning tree

is equivalent to the facility location problem. (Indeed, it is this connection to the facility location problem that was used in the work on compressing tables of data mentioned earlier [5].) In the terminology of [15], each page is a possible facility; a page that is not compressed by a reference corresponds to an opened facility; and a page that is compressed using a reference corresponds to a location receiving shipment from a facility corresponding to the reference page. We believe the depth-bounded directed minimum spanning tree problem is an interesting extension of previous facility location problems. In practice, we expect that using the FIND-REFERENCE algorithm to initially find a directed tree and then "chopping the tree" to maintain a depth bound (by changing some nodes to be compressed without a reference and thus linking them to the root r) is a suitable solution.

4 Hardness results

Since we can find the optimal compression given an appropriate cost metric when we allow a single reference node using branching algorithms, a natural question to ask is whether we can similarly achieve optimality when we allow more than one reference node. We show hardness results related to this question. We focus on the case where up to two nodes can be used as references, but everything described is easily generalized to any number of reference nodes.

For up to two reference nodes, the affinity graph becomes the following kind of structure:

Definition 1 *A* 2-supergraph *is a directed hypergraph where each hyperedge is directed from a single node to two other nodes. These two other nodes can be the same, but must be different from the source node.*

Given a Web graph, we shall consider the corresponding weighted 2-supergraph, where w_{ijk}, the weight of the hyperedge from i to j and k, represents the cost of encoding i using both j and k as references. For a hyperedge w_{ijj} where the two other nodes pointed to are the same node, the weight of the hyperedge w_{ijj} represents the cost of encoding node i using only node j as a reference node. Note that w_{ijk} will vary depending on the overlap between the set of edges of the Web graph that nodes i and j have in common and the set of edges of the Web graph that nodes i and k have in common. We call the resulting 2-supergraph an *affinity 2-supergraph.*

Given a Web graph, computing the affinity 2-supergraph for a given link compression scheme can easily be done in polynomial time. Using the affinity 2-supergraph to compute the best compression using up to two reference nodes is equivalent to the following generalization of finding optimal branchings:

Definition 2 *Given a 2-supergraph $\mathcal{G}$ and a designated root node r, a* 2-branching *is a subset S of the hyperedges of $\mathcal{G}$ such that each node except the designated root has exactly one outgoing hyperedge in S, and r has no outgoing hyperedges in S. In addition, the hypergraph formed by the set of hyperedges in S has no directed cycles. The optimum 2-branching is the 2-branching that minimizes the total weight of the edges in S.*

Unfortunately, in general finding the optimal 2-branching is not only NP-Hard, it is hard to approximate. In particular, we demonstrate an approximation preserving polynomial time reduction from the problem of finding the optimal directed Steiner tree to the problem of finding the optimal 2-branching. It is known that if $P \neq NP$, then no polynomial time algorithm can find a $\log n$-approximation to the directed Steiner tree problem [7]. Furthermore, this reduction provides evidence that it may be difficult to find a polynomial time algorithm that provides better than an n^{ϵ}-approximation, since the directed Steiner tree problem has thus far resisted efforts to outperform this bound.

Theorem 2 *Any polynomial algorithm that provides a k-approximation for the 2-branching problem also provides a k-approximation for the directed Steiner tree problem.*

Proof: We use the following reduction: given an input to the directed Steiner tree problem consisting of a directed graph $G = (V, E)$, a set $S \subseteq V$ of required points, a weight function $W : E \rightarrow \mathcal{R}$, and a designated root r, we construct an input to the 2-branching problem as follows. The input hypergraph is $G' = (V', E')$, where $V' = V \cup U$, and U is an additional set of nodes we describe. For each node $v \in V - S$, let $U(v) = \{u_1, \ldots, u_{indeg(v)}\}$ be the set of nodes $u \in V$ such that $(u, v) \in E$. For each $u_i \in U(v)$, U has a node denoted v_i. Let $u(v_i)$ be the node u_i that corresponds to the node v_i.

For each edge $e = (u, v)$ in E, E' contains one hyperedge $e' = (u, v, v)$, where w_{uvv} is equal to the weight of e. The edge set E' also contains a hyperedge $(v, v_1, u(v_1))$ of weight 0, and for all i, $1 \leq i < indeg(v)$, E' contains a hyperedge $(v_i, v_{i+1}, u(v_{i+1}))$ of weight 0. The edge set E' also has a hyperedge $(v_{indeg(v)}, r, r)$ of weight 0. Finally, for all i, $1 \leq i \leq indeg(v)$, E' has a hyperedge (v_i, v, r) of weight 0. This completes the construction of the 2-branching problem. The theorem now follows from the following two claims:

Claim 1 *For any directed Steiner tree of weight z, there is a 2-branching with weight z.*

Proof: The 2-branching consists of all the edges used by the directed Steiner tree. These will be the only non-zero weight edges used in the 2-branching, and thus the 2-branching has the same weight as the directed Steiner tree. This guarantees that every required node of the Steiner tree problem has exactly one outgoing hyperedge. We divide the optional nodes of the Steiner tree problem into two sets: S_1, consisting of the nodes used in the Steiner tree solution, and S_2, consisting of the nodes not used in the Steiner tree solution. For each node $v \in S_1$, there already is exactly one outgoing hyperedge for v, and thus we only need to specify the outgoing hyperedges for the nodes v_i. For each such v_i, we include the hyperedge (v_i, v, r). Since there are no hyperedges pointing at the nodes v_i, this will not create any cycles.

For each node $v \in S_2$, we include the hyperedge $(v, v_1, u(v_1))$, and for all i, $1 \leq i < indeg(v)$, we include the hyperedge $(v_i, v_{i+1}, u(v_{i+1}))$. We also include the hyperedge $(v_{indeg(v)}, r, r)$. Since there are no edges in the Steiner tree solution coming into or going out of v, this will not create any cycles. We have specified an outgoing hyperedge for every node without introducing any cycles, and thus we have a valid 2-branching. $\Box$

Claim 2 *For any 2-branching of weight z, there is a directed Steiner tree with weight z.*

Proof: The 2-branching must consist of exactly one hyperedge from each of the required Steiner nodes. These hyperedges correspond exactly to edges in the Steiner tree problem, and thus these edges are included in the Steiner tree solution. For each of the optional Steiner nodes v, the hyperedge from v either points to another node in the original Steiner tree problem, or it uses the hyperedge $(v, v_1, u(v_1))$. In the first case, we include that edge and the node v in the Steiner tree solution, and since the 2-branching has no cycles, this cannot create any cycles in the Steiner tree solution. In the second case, it is easy to show by induction that the 2-branching must also contain all of the hyperedges $(v_i, v_{i+1}, u(v_{i+1}))$. Since there can be no cycles in the 2-branching, the only nodes with hyperedges pointing to v are the nodes v_i. Thus, we can leave the node v out of the Steiner tree solution. This gives us a valid solution to the Steiner tree problem where the set of nonzero weight edges of the Steiner tree solution is exactly to the set of nonzero weight hyperedges of the 2-branching. □

The inapproximability result above demonstrates the hardness of the general problem of finding an optimal 2-branching. This result does not however directly imply that it is even NP-Hard to find the best compression using at most two references, since the graphs that we reduce the directed Steiner tree problem to may not correspond to actual affinity graphs that arise as a result of a Web graph.

We also provide a more direct reduction showing that it is in fact NP-Hard to find the best compression of a Web graph based on using up to two reference nodes. In fact, even if we ignore the additional difficulty imposed by taking into account the asymmetry of the affinity graph, the problem remains NP-Hard. In particular, we demonstrate that the problem of finding the assignment of reference nodes that maximizes the total number of edges in the Web graph that are represented by a corresponding edge in a reference node is NP-Hard. This proof can easily be extended to the case where the objective is to minimize the total cost of the reference nodes used.

Theorem 3 *The problem of finding an encoding for a graph G_W, with each node encoded using up to two reference nodes, that maximizes the total number of edges that are encoded using a reference node is NP-Hard.*

The proof, which is somewhat lengthy, is a reduction from 3SAT, and is given in the Appendix. Note that this does not imply that the problem of maximizing the number of edges encoded using a reference node is hard to approximate. In fact, it is clear that we can find a 2-approximation to this problem by using at most a single reference node to encode each node. Achieving better than a 2-approximation is an interesting open problem.

5 Experiments

We present the results from a preliminary prototype running on artificial random copy graphs and one subset of a snapshot of the Web graph. We emphasize that these experiments are meant as a preliminary proof of concept. In particular, the prototype does not output a compressed file, but rather the compressed size of the file. Moreover, when using Huffman coding, the compressed size does not include the size of any associated Huffman tables; we

chose not to include this as the size of the Huffman table depends on whether one compresses it further.

We first describe the graphs we tested. For the random copy graphs, our tests all had 131072 nodes. (Smaller test graphs had similar performance, so we present results for the largest graphs we tested.) Each graph began with 1024 seed nodes with three outedges, where the end of the outedge was chosen uniformly at random from all nodes. When new nodes were added, they were given only outedges. The outedges were determined by copying edges from some number of nodes and by generating edges with endpoints chosen uniformly at random from all present nodes. We show the parameters for the copy graphs tested in Table 1. The field # random copies denotes the number of nodes whose outedges were copied. A range such as $[1, 2]$ denotes that an integer value was chosen uniformly over that range. Each edge was copied with a fixed probability, listed as the copy probability. The field # random edges gives the number of edges that were generated with random destinations; again, a range denotes that an integer value was chosen uniformly over that range. We note that for the large graphs G_3 and G_4, we were forced by memory considerations to limit the affinity graph to allow edges between nodes i and j only if their outedges share at least three destinations. This can only hurt our compression efforts. (Further testing suggests that the the difference is minimal if two shared destinations can be handled in constructing the affinity graph.)

We also tested our compression scheme on real Web data from from the TREC-8 (Text REtrieval Conference 8) Web track [9]. Our data set was the WT2g data set[3], which was chosen as a small subset of the Web for testing information by the TREC retrieval conference. This data set is larger than our random sets; hence again to construct the affinity graph on the TREC database we only created edges between pages with at least three shared links.

Table 2 presents the compression results. Here we have taken the average of ten different trials for the random graphs, where a different random graph is produced for each trial. We note that there is little deviation between the runs. The average number of edges is given; the uncompressed size given is simply the number of edges multiplied by $log_2(\#\text{nodes})$, which is an underestimate of the uncompressed size. Compression for other methods is given as a percentage of the uncompressed size.

As seen in graph G_1, when the amount of copying is low, and thus the average degree is very small, the reference algorithm alone does slightly worse than the Huffman algorithm, although using a Huffman code in conjunction with the reference algorithm leads to better performance. When the amount of copying is larger, as for G_2, G_3, and G_4, our Find-Reference algorithm greatly outperforms Huffman coding. We expect repeated passes might allow even greater compression. The Huffman algorithm compresses the outedges for each edge, so the code words are based on the indegree. For the Find-Reference algorithm, we test both the straightforward algorithm as well as one which first determines the references and then uses Huffman coding on the remaining outedges.

Our results are actually best for the TREC database, demonstrating that our approach should be effective on real Web data as well. Our belief is that our good results on the TREC data set arise because links appear to have significant locality, due to the heuristic principles by which the data set was chosen [9]. We believe much greater testing is required

[3] Our copy has sixty-three fewer pages than stated in [9]; we are not aware of the reason for the discrepancy.

to determine our performance on larger scale Web data sets, although this preliminary result is promising.

Name	G_1	G_2	G_3	G_4	TREC
Nodes	131072	131072	131072	131072	247428
# Random Copies	1	1	[1,2]	[0,4]	NA
Copy prob.	0.5	0.7	0.5	0.5	NA
# Random Edges	1	1	[1,2]	[1,2]	NA

Table 1: Parameters of the test graphs.

Name	G_1	G_2	G_3	G_4	TREC
Edges	273787.3	426294.6	668338.8	1339779	1166702
Uncompressed	4654384.1	7247008.2.1	11361759.6	22776243	21000636
Huffman	87.75	83.93	85.15	79.47	83.31
Find-Reference	88.68	67.49	69.96	61.65	49.15
F.Ref. + Huff.	81.58	63.63	65.35	54.13	46.36

Table 2: Results from the test graphs.

6 Future Work

We have initiated study into how to compress Web graphs. We have considered the copy graph model, introduced elsewhere as a random graph family with properties similar to Web graphs. Using this structure, we have designed a compression algorithm based on finding similarity among the links of the pages and tested it on simple copy graphs. We have also shown that various generalizations of this idea lead to NP-Hard problems.

There are several directions remaining to pursue, including further tests of our algorithm on real Web data. It would also be interesting to learn how our approach works in conjunction with others. For example, another idea that would clearly be useful in compressing Web graphs is locality. If we assume that we store our graph with pages listed alphabetically by URL, we would expect a good percentage of the links to be between pages that are near each other in the list, since pages in the same domain are likely to reference each other. Thus, even before using our reference-based algorithm, it seems likely that a first pass algorithm to handle local links would be useful. One natural approach would be to split the Web graph into two subgraphs, one with local links (say, within distance 256 in the sorted URL list) and non-local links and compress them separately. Although designing such a system can be done via experimentation, developing an appropriate model that allows us to understand the tradeoffs would be an interesting problem.

Another interesting issue is understanding what our compression algorithm tells us about the structure of graphs. For example, a natural technique to test how accurately a proposed random graph model captures the structure of real Web graphs would be to run our compression algorithm (or any other compression algorithm) on both kinds of graphs, and compare

the compression obtained. A feature more specific to our algorithm is that our algorithm attempts to reconstruct the evolution of a copy graph when choosing reference nodes for each edge. Preliminary results seem to indicate that the algorithm is able to correctly identify a large fraction of the history, and thus an interesting line of research would be to quatify this, and to understand its implications for real Web graphs.

References

[1] W. Aiello, F. Chung, and L. Lu. A random graph model for massive graphs. In *Proceedings of the 32nd Annual ACM Symposium on Theory of Computing*, pages 171-180, 2000.

[2] K. Bharat, A. Broder, M. Henzinger, P. Kumar, and S. Venkatasubramanian. The Connectivity Server: fast access to linkage information on the Web. In *Proceedings of the 7th World Wide Web Conference*, 1998. Available at http://www7.scu.edu.au/programme/fullpapers/1938/com1938.htm.

[3] S. Brin and L. Page. The anatomy of a large-scale hypertextual Web search engine. In *Proceedings of the 7th World Wide Web Conference*, 1998. Available at http://www7.scu.edu.au/programme/fullpapers/1921/com1921.htm.

[4] A. Broder, R. Kumar, F. Maghoul, P. Raghavan, S. Rajagopalan, R. Stata, A. Tomkins, and J. Wiener. Graph structure in the Web: experiments and models. In *Proceedings of the 9th World Wide Web Conference*, 2000. Available at http://www9.org/w9cdrom/index.html.

[5] A. L. Buchsbaum, D. F. Caldwell, K. W. Church, G. S. Fowler, and S. Muthukrishnan. Engineering the compression of massive tables: an experimental approach. In *Proceedings of 11th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 175-184, 2000.

[6] P. M. Camerini, L. Fratta, and F. Maffioli. A note on finding optimum branchings. *Networks*, 9:309-312, 1979.

[7] M. Charikar, C. Chekuri, T. Cheung, Z. Dai, A. Goel, S. Guha, and M. Li. Approximation algorithms for directed Steiner problems. In *Proceedings of the Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 192-200, San Francisco, California, 25-27 January 1998.

[8] H. N. Gabow, Z. Galil, T. Spencer, and R. E. Tarjan. Efficient algorithms for finding minimum spanning trees in undirected and directed graphs. *Combinatorica* 6(2):109-122, 1986.

[9] D. Hawking, E. Voorhees, N. Craswell, and P. Bailey. Overview of the TREC-8 Web Track. In *The 8th Text Retrieval Conference*, 2000. Available at `http://trec.nist.gov/pubs/trec8/t8_proceedings.html`.

[10] R. M. Karp. A simple derivation of Edmonds' algorithm for optimum branchings. *Networks*, 1:265-272, 1972.

[11] J. Kleinberg. Authoritative sources in a hyperlinked environment. In *Proceedings of 9th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 668-677, 1998.

[12] J. Kleinberg, R. Kumar, P. Raghavan, S. Rajagopalan, and A. Tomkins. The Web as a graph: Measurements, Models, and Methods. In *Proceedings of the International Conference on Combinatorics and Computing*, 1999.

[13] R. Kumar, P. Raghavan, S. Rajagopalan, and A. Tomkins. Trawling the Web for emerging cybercommunities. In *Proceedings of the 8th World Wide Web Conference*, 1999.

[14] R. Kumar, P. Raghavan, S. Rajagopalan, and A. Tomkins. Extracting large scale knowledge bases from the Web. In *Proceedings of the 25th VLDB Conference*, 1999.

[15] D. B. Shmoys, E. Tardos, and K. Aardal. Approximation algorithms for facility location problems. In *Proceedings of the 29th ACM Symposium on Theory of Computing*, pages 265-274, 1997.

[16] R. E. Tarjan. Finding Optimum Branchings. *Networks*, 7:25-35, 1977.

Appendix 1

In this section we give a brief asymptotic analysis of the Huffman-based compression scheme described in Section 2 using some simplifying assumptions. The argument we present can be made more rigorous by using concentration bounds; for simplicity, we work with expectations.

If we ignore the stop and end of file symbols, we may obtain an approximate expression for the asymptotic average encoding length by computing the entropy per edge. Let us assume that the fraction of pages with indegree j is proportional to $1/j^\alpha$ with $\alpha > 2$; note in particular we simplify by assuming no node has indegree 0. Let $\zeta(\alpha) = \sum_{j=1}^{\infty} 1/j^\alpha$. The probability that a node with non-zero indegree has indegree j is then $1/\left(j^\alpha \sum_{k=1}^{\infty} 1/k^\alpha\right) = 1/\left(j^\alpha \zeta(\alpha)\right)$. Hence the expected number of total edges is

$$\frac{n \sum_{j=1}^{\infty} j/j^\alpha}{\sum_{j=1}^{\infty} 1/j^\alpha} = \frac{n\zeta(\alpha-1)}{\zeta(\alpha)},$$

where n is the number of nodes in the graph. Ignoring the problem of fractional length symbols, the length of a Huffman symbol for a node that has indegree j would be chosen as $\log \frac{n\zeta(\alpha-1)}{j\zeta(\alpha)}$. The expected number of nodes with indegree j is $n/\zeta(\alpha)j^\alpha$. Hence, assuming that the number of nodes with indegree j equals its expectation, the total number of bits used is

$$\sum_{j=1}^{\infty} \frac{n}{\zeta(\alpha)j^\alpha} \cdot j \log \frac{n\zeta(\alpha-1)}{j\zeta(\alpha)} = \frac{n\zeta(\alpha-1)}{\zeta(\alpha)} \log \frac{n\zeta(\alpha-1)}{\zeta(\alpha)} - \frac{n}{\zeta(\alpha)} \sum_{j=1}^{\infty} \frac{\log j}{j^{\alpha-1}}.$$

With $\alpha > 2$, the above expression is $\Omega(m \log n)$, where m is the number of edges in the graph. Hence, this kind of compression requires asymptotically the same number of bits as not using Huffman compression.

Appendix 2

In this appendix we provide the proof of Theorem 3, restated here for convenience:

Theorem 3 *The problem of finding an encoding for a graph G_W, with each node encoded using up to two reference nodes, that maximizes the total number of edges that are encoded using a reference node is NP-Hard.*

Proof: We describe a process such that given a 3SAT formula Φ, constructs a web graph $G_W(\Phi)$, and a value $K(\Phi)$, such that $G_W(\Phi)$ has an encoding using two reference nodes per node which encodes $K(\Phi)$ pointers using the reference nodes if and only if Φ is satisfiable.

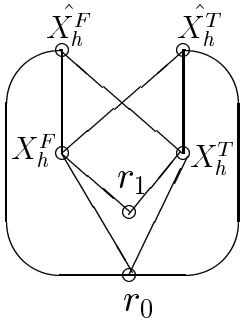
Given any web graph G_W, let the *simplified affinity* graph G_S corresponding to G_W be a weighted graph such that the nodes of G_S are the same as the nodes of G_W, and $w(n_1, n_2)$, the weight of the edge between n_1 and n_2 in G_S, is the number of nodes that are pointed to by both n_1 and n_2 in G_W. Note that this implies that G_S is an undirected graph. Since we are interested in compression using two reference nodes, another quantity of interest is $w(n_1, n_2, n_3)$, which can be thought of as the weight of the (undirected) hyperedge between n_1, n_2, and n_3, and is defined to be the number of nodes that are pointed to by all of n_1, n_2 and n_3 in G_W. Note that if n_1 is encoded using n_2 and n_3 as reference nodes, then the number of pointers that are encoded using the reference nodes is $w(n_1, n_2) + w(n_1, n_3) - w(n_1, n_2, n_3)$. Thus, given a simplified affinity graph, it is trivial to compute the corresponding affinity 2-supergraph.

In our reduction from 3SAT, we describe the web graph $G_W(\Phi)$ by first describing a process of mapping Φ to a simplified affinity graph $G_S(\Phi)$ such that $G_S(\Phi)$ has a 2-branching of weight $K(\Phi)$ if and only if Φ is satisfiable. We then demonstrate how to construct the web graph $G_W(\Phi)$ such that the simplified affinity graph corresponding to $G_W(\Phi)$ is $G_S(\Phi)$. This will complete the proof of the theorem.

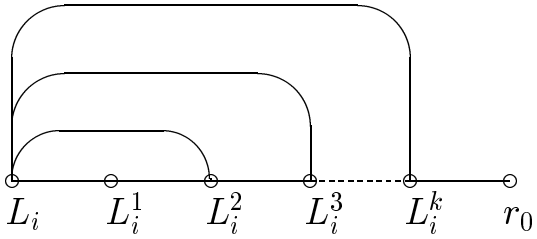
In the graph $G_S(\Phi)$, the maximum weight of any edge is Δ, where the value of Δ will be set below. Let the set of edges that have this weight be called the set of *full* edges. We refer to any pair of edges (n_1, n_2) and (n_1, n_3) where $w(n_1, n_2, n_3) = 0$ as a *non-overlapping* pair. If $w(n_1, n_2, n_3) = 1$, then (n_1, n_2) and (n_1, n_3) are a *lightly-overlapping* pair, and if $w(n_1, n_2, n_3) > 1$, then (n_1, n_2) and (n_1, n_3) are a *fully-overlapping*, or *overlapping* pair.

We next describe how to construct $G_S(\Phi)$ from Φ. In what follows, we shall describe the edges of $G_S(\Phi)$ in terms of its full edges; all other edges have weight less than Δ. We also assume that the 2-branching problem is defined in such a manner that there is a given node r_0 that is required to be the node that does not use any reference nodes, and another given node r_1 that is required to use at most one reference node. Removing this assumption is not difficult.

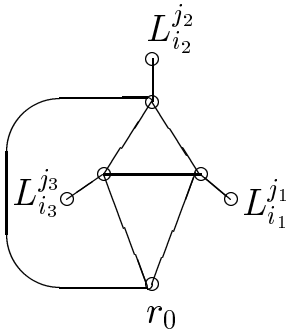
For each variable X_h in the formula Φ, the graph $G_S(\Phi)$ has a copy of the following structure:



The node X_h^T is connected to the clause structures that the literal X_h appears in, and X_h^F is connected to the clause structures that $\bar{X}_h$ appears in. In particular, let L_i be one of the nodes X_h^T or X_h^F. The graph $G_S(\Phi)$ has a node L_i^j for the jth appearance of the literal L_i in a clause. We use L_i^j to denote both the appearance of the literal in the formula, as well as the corresponding node in the graph. The nodes are connected as follows, where k_i is the number of appearances of L_i:



There is also a structure for each clause. In particular, a clause containing the three literals $L_{i_1}^{j_1}$, $L_{i_2}^{j_2}$, and $L_{i_3}^{j_3}$ results in the following structure:



The final edge we add to the graph is (r_0, r_1). These are all of the full edges of the graph $G_S(\Phi)$. To complete the description, we need to specify which pairs of edges are overlapping. All of the pairs of edges incident to L_i are overlapping, except for the pair of edges $(L_i, \hat{L_i}), (L_i, L_i^1)$, which is non-overlapping and the pair $(L_i, r_0), (L_i, r_1)$, which is lightly-overlapping. Of the three pairs of edges incident to $\hat{L_i}$, two are overlapping: the non-overlapping pair is $(\hat{L_i}, \bar{L_i}), (\hat{L_i}, r_0)$, where $\bar{L_i}$ is the complement of L_i. All other pairs of edges are non-overlapping.

This construction is easily seen to be polynomial time. We see that it is a reduction from the following two claims.

Claim 3 *If Φ is satisfiable, then $G_S(\Phi)$ has a 2-branching of weight $(2N-3)\Delta - v$, where N is the total number of nodes in the graph, and v is the number of variables in Φ.*

Proof: We describe how to construct such a 2-branching B, given a satisfying assignment. For each true literal L_i, B contains the (directed) edges (L_i, r_0), (L_i, r_1), $(\hat{L_i}, \bar{L_i})$, and $(\hat{L_i}, r_0)$. For every false literal $\bar{L_i}$, B contains the edges $(\bar{L_i}, \bar{L_i^1})$, $(\bar{L_i}, \hat{\bar{L}}_i)$, $(\hat{\bar{L}}_i, L_i)$, and $(\hat{\bar{L}}_i, r_0)$. In addition, B contains the edges $(L_i^1, L_i), (L_i^1, L_i^2), (L_i^2, L_i), (L_i^2, L_i^3), \ldots, (L_i^{k_i}, L_i), (L_i^{k_i}, r_0)$, as well as the edges $(\bar{L_i^1}, \bar{L_i^2}), (\bar{L_i^2}, \bar{L_i^3}), \ldots, (\bar{L_i^{k_i}}, r_0)$, plus the edge from each node $\bar{L}_i^j$ to the corresponding clause structure.

It is easy to verify that in the description of B thus far there are no cycles and every node other than the three in each clause structure has exactly two outgoing edges. The 2-branching B also contains an edge from each of the three nodes in each clause structure to the node r_0. Finally, we need to specify the second outgoing edge from each of these three nodes. However, each clause has at least one true literal, and thus there is at least one node $L_{i_c}^{j_c}$ such that the edge from $L_{i_c}^{j_c}$ to the clause structure was not included in B. We include the edge from the clause structure to such a node $L_{i_c}^{j_c}$. The outgoing edges in B from the other two nodes in the clause structure are the clockwise edges along the triangle of the clause structure. These additional edges mean that every node has exactly two outgoing edges, and we have guaranteed that we do not have a cycle going around the clause structure.

Before adding the edge from the clause structure to the node $L_{i_1}^{j_1}$, it was easy to verify that there were no cycles, since there were no directed paths between the nodes corresponding to different variables. However, this additional edge does introduce such paths. To see that these edges do not introduce any cycles, note that they can only introduce paths from an L_i representing a false literal to a L_j representing a true literal. It is also possible to go from the node L_i to the node $\bar{L}_i$, but only in the case that L_i is the false literal. Thus, any path that goes from a node L_i to a node L_j (which may be $\bar{L}_i$) must go from a false literal to a true literal. Therefore, there can be no cycles in the resulting set of edges B. The final edge we add is (r_1, r_0). Since r_0 has no outgoing edges, this does not create a cycle.

Thus, B is a valid 2-branching. Every node except r_0 and r_1 has two outgoing full edges in B, and for every node except the nodes L_i corresponding to true literals, these two edges are non-overlapping. The two outgoing edges from the L_i are lightly overlapping. Since we have exactly v true literals, the total weight of this 2-branching is $(2N-3)\Delta - v$. $\square$

Claim 4 *If $G_S(\Phi)$ has a 2-branching of weight $(2N-3)\Delta - v$, then Φ is satisfiable.*

Proof: For any variable X_i, if all four of the nodes in the variable structure for X_i had two full, non-overlapping edges, this would lead to a cycle. Thus, for each of the v variable structures, there is at least one node whose outgoing edges contribute at most $2\Delta - 1$ to the total weight of the 2-branching. In order to have a 2-branching of weight $(2N-3)\Delta - v$, there must be exactly one such node per variable structure, and it must contribute $2\Delta - 1$ to the branching. The only way for this to be possible is if there is one literal L_i that has the edges (L_i, r_0), (L_i, r_1), and the other literal $\bar{L}_i$ has the edges $(\bar{L}_i, \bar{L_i^1})$ and $(\bar{L}_i, \hat{\bar{L}}_i)$. In order to satisfy the formula Φ, it is sufficient to set the literal L_i to true, and the literal $\bar{L}_i$ to false.

To see that this produces a true literal in every clause, note that every other node in $G_S(\Phi)$ must have two full and non-overlapping outgoing edges, and consider any clause structure C. For C to not have a cycle around the triangle, and every node in C to have two outgoing edges, there must be some edge in the 2-branching from C to some node L_i^j, corresponding to a literal in the clause represented by C. However, if the edge (L_i, L_i^1) is in the 2-branching, and every node has two outgoing edges, then each L_i^j must have an edge to its corresponding clause structure. Thus, the edge from C to L_i^j implies that the edge (L_i, L_i^1) is not in the 2-branching, and thus L_i must be set to true in Φ. Thus, Φ is satisfied by our assignment. $\square$

Finally, we describe how to construct $G_W(\Phi)$ from $G_S(\Phi)$. It is sufficient that the construction of $G_W(\Phi)$ preserves the following aspects of our definition of $G_S(\Phi)$:

- All described full edges have weight Δ.
- All other edges have weight at most $\Delta/2$.
- All pairs of full edges are overlapping, lightly-overlapping, or non-overlapping, as described.

Note note that for any pair of edges such that at least one of the edges is not full, it does not matter if that pair is overlapping or not. In the graph $G_W(\Phi)$, there is a node for every node of $G_S(\Phi)$. In addition, $G_W(\Phi)$ has a set of nodes V_1 which do not have any outgoing edges. The weights of the edges of $G_S(\Phi)$ will be achieved by placing edges from the nodes in $G_W(\Phi)$ corresponding to the nodes of $G_S(\Phi)$ to the nodes in V_1. Since the nodes in V_1 have no outgoing edges, the weight of all the edges incident to those nodes in the graph $G_S(\Phi)$ will be 0, and thus the nodes in V_1 can be ignored in the graph $G_S(\Phi)$.

Let d be the maximum degree of any node in $G_S(\Phi)$. We set $\Delta = 4d$. For every full edge (n_1, n_2) in $G_S(\Phi)$, n_1 and n_2 in $G_W(\Phi)$ have edges to exactly Δ of the same nodes in V'. Call this set of nodes in V' the *target* of (n_1, n_2). There are no other edges in $G_W(\Phi)$, and to start with, the targets of each full edge in $G_S(\Phi)$ are disjoint. This gives us a graph $G'_W(\Phi)$ such that the corresponding $G'_S(\Phi)$ has the correct set of full edges, each with weight Δ, and all other edges have weight 0. However, all pairs of edges are non-overlapping.

To make some of the edges overlapping, we shall merge some pairs of nodes in V' into a single node (with edges to it from all of the places where the original two nodes had pointers to them). In particular, for every pair of edges (n_1, n_2) and (n_1, n_3) in $G_S(\Phi)$ that is overlapping, we merge two nodes from the target of (n_1, n_2) with two corresponding nodes in the target of (n_1, n_3) (resulting in two new nodes). The two merged nodes in the target of an edge (n_1, n_2) are different for every edge that (n_1, n_2) is merged with. We apply the same process with all lightly-overlapping pairs of edges, with the difference that only one pair of nodes is merged. Note that the edge (n_1, n_2) has at most $d-1$ overlapping or lightly-overlapping edges incident to each of n_1 and n_2, and thus after the merger, there will still be at least $2d + 2$ nodes that haven't been merged in the target of (n_1, n_2).

Consider the merger defined by a pair of overlapping (full) edges (n_1, n_2) and (n_1, n_3). No node other than n_1, n_2 and n_3 points to nodes in the targets of (n_1, n_2) and (n_1, n_3), and thus this merger cannot effect the weight of any edge in $W_S(\Phi)$ other than (n_1, n_2), (n_1, n_3),

or (n_2, n_3). Furthermore, since the targets of these two edges are distinct, the merger does not effect the weight of (n_1, n_2) or (n_1, n_3), but it does ensure that those two edges become overlapping.

The only edge weight that is changed in $W_S(\Phi)$ by the merger is (n_2, n_3), which is increased by either 1 or 2, depending on whether the merger resulted from an overlapping pair or a lightly-overlapping pair. If (n_2, n_3) is a full edge, then this increase is offset by removing one or two (depending on the increase) of the nodes from the target of (n_2, n_3) that were not involved in any mergers. Otherwise, no further changes are required. Since the only pairs of edges that can increase the weight of (n_2, n_3) via a merger are pairs of full edges of the form (n_i, n_2) and (n_i, n_3), for some i, and since the maximum degree of $G_S(\Phi)$ is d, we see that the weight of any edge (n_2, n_3) can be increased by at most d mergers. Thus, no edge that was not full increases in weight past $2d = \Delta/2$, and each edge that is full has enough nodes left in its target after its mergers to account for the effects of all other mergers, and thus remains at weight Δ.

Thus, we have constructed a graph $G_W(\Phi)$ such that the corresponding graph $G_S(\Phi)$ satisfies all three of our sufficient conditions. Determining the maximum number of nodes that can be encoded using reference nodes in an encoding of $G_W(\Phi)$ using two reference nodes per node is equivalent to deciding if the formula Φ is satisfiable. This completes the reduction, and the proof. $\square$