Technical Report

Department of Computer Science
and Engineering
University of Minnesota
4-192 Keller Hall
200 Union Street SE
Minneapolis, MN 55455-0159 USA

TR 14-018

Storing Dynamic Graphs: Speed vs. Storage Trade-offs

Jeremy Iverson and George Karypis

July 30, 2014

Storing Dynamic Graphs: Speed vs. Storage Trade-offs

Submitted for Blind Review

***Abstract*—With the ever increasing size and availability of web and social graph comes the need for compact and efficient data structures in which to store them. The problem of compact representations for sparse static graphs is a well studied problem in the domain of web and online social graphs, namely the WebGraph Framework and the Layered Label Propagation ordering for extending WebGraph to online social networks, among others. While these techniques do a satisfactory job in the context of static sparse graphs, there is little literature on how these techniques can be extended to dynamic sparse graphs. In this paper, we present algorithms and experimental analysis for five data structures for representing dynamic sparse graphs: Linked-List (LL), Batch Compressed Sparse Row (BCSR), Dynamic Adjacency Array (DAA), Dynamic Intervalized Adjacency Array (DIAA), and Dynamic Compressed Adjacency Array (DCAA). The goal of the presented data structures is two fold. First, the data structures must be compact, as the size of the graphs being operated on continues to grow to less manageable sizes. Second, the cost of operating on the data structures must be within a small factor of the cost of operating on the static graph, else these data structures will not be useful. Of these five algorithms, LL, BCSR, and DAA are baseline approaches, DIAA is semi-compact, but suited for fast operation, and DCAA is focused on compactness and is a dynamic extension of the WebGraph Framework. Our results show that for well intervalized graphs, like web graphs, DIAA is superior to all other data structures in terms of memory and access time. Furthermore, we show that in terms of memory, the compact data structure DCAA outperforms all other data structures at the cost of a modest increase in update and access time.**

***Keywords*—dynamic, sparse, graph, network, data structures, compression, web graph, social graph**

I. Introduction

With the staggering growth rate of web graphs and online social network datasets in the past decade, and the increasing demand for more efficient query and analysis of these networks, comes a renewed interest in the data structures and algorithms with which we represent and explore these networks. Historically, researchers have been limited to studying static snapshots of the networks off-line. However, this simplistic approach has two major drawbacks. First, real-time analysis is not possible when the network in question must be captured and output before analysis can commence. Second, the exploration of static networks is inherently limited by their time independent nature. It has been shown that the evolution of networks can illuminate interesting and important insights about the structure of the network [1].

Recently, researchers have acknowledged these limitations of static network analysis and much of the focus has shifted to dynamic network analysis. However, many of the contributions which can be attributed to this shift are aimed at specific problem instances. For example, there is a great deal of literature related to the data structures and algorithms necessary for maintaining network connectivity [2], shortest paths [3], or maximal cliques [4], but there is a dearth of research addressing the problem of which data structures are most appropriate for a general dynamic network framework on top of which more complex analysis can be performed.

There is a critical trade-off to consider when choosing the appropriate data structure to solve this problem. That trade-off being the amount of memory consumed to store the data structure versus the efficiency of operating on the data structure. Considering the size of web and social networks, this can be a challenge, since data structures which do not fit in memory will be more expensive to operate on, but compressing networks to fit in memory comes at the price of compression and decompression overhead. In the context of static networks, this is a well studied and understood problem. An abundance of research can be found on compact network representations [5], [6], [7], as well as access efficient data structures [8]. In some cases, the work done on static networks can be directly extended to the dynamic context, while in other cases it cannot.

In this paper we investigate the space of data structures for representing dynamic networks in terms of computational efficiency for network analysis queries, as well as memory efficiency. The latter is an important factor, since the dynamic networks which are the focus of this paper are web and online social networks, both of which are increasingly large. Along with previously known dynamic network data structures like: Linked-List (LL), Batch Compressed Sparse Row (BCSR), and Dynamic Adjacency Array (DAA), we introduce two extensions of static network representations: Dynamic Intervalized Adjacency Array (DIAA) and Dynamic Compressed Adjacency Array (DCAA), which are especially applicable to the types of networks addressed by this paper. We theoretically and experimentally evaluate each of the data structures in terms of operation time and memory requirements. Over our benchmark suite, we show that in terms of operation time, BCSR is superior to the other data structures and LL is inferior in all cases. We also show that BCSR requires more memory under all circumstances and does not scale as well as DIAA. In terms of memory, DCAA outperforms all other data structures, but in most cases, does so at the cost of operation time. DAA and DIAA strike a balance between memory requirements and operation time, with DIAA exceeding DAA in operation time, as well as memory, for well intervalized graphs, such as web graphs.

In Section II we introduce and define necessary concepts and notation. Section III presents the current state-of-the-art data structures for storage of static web and social graphs. Then, in Section IV, the problem which this paper addresses, is

formally defined and in Section V, our proposed data structures to solve this problem described in detail along with theoretical analysis of their memory requirements and update and access complexity. Section VI outlines relevant details of the data structure implementation and the experimental procedures. This is followed by Section VII which is a in-depth discussion of the obtain experimental results. The paper is concluded with Section VIII, which summarizes our proposed data structures and the conclusions drawn from the experimental results.

II. Definitions and Notation

A graph $G = (V, E)$ consists of a set of vertices V and a set of edges E. The set of adjacent vertices of a particular vertex u is expressed as $\Gamma(u)$ and the degree of u can then be stated as $\delta(u) = |\Gamma(u)|$. The maximum degree of any vertex in G is denoted $\Delta(G) = \max_{\forall u \in V} \delta(u)$ and by abuse of notation, simply Δ. In terms of graph data structures, *dynamic context* refers to the scenario in which the data structure used to store the graph is persisted across all *updates*: addition and removal of edges or vertices, to the graph.

A *web graph* is a directed graph whose vertex set V, represents pages from the World Wide Web and whose edge set E, captures the hyperlinks between the pages which exist in V. An *online social graph* is a directed or undirected graph where vertices represent individuals or organizations and the edges imply some type of relationship between the participating entities. Classic examples of online social graphs include Twitter (directed) and Facebook (undirected). In the discussion that follows, it will be assumed that the graphs are undirected, since storage of directed graphs can be directly extended from storage of undirected graphs by storing an indicator of directedness with each edge or storing the graph transpose.

III. Static Graph Representations

There are two commonly used data structures for static sparse graphs. The first is vertex-centric and is typically referred to as adjacency list representation. In this representation, each vertex is treated as an object and stores a list of the vertex ids of those vertices which it is adjacent to. In its most standard form, this representation uses linked-lists or arrays to store each vertex' adjacency list. However, the *compressed sparse row (CSR)* or *Yale format*, can also be used to store vertex adjacency lists in a more compact / cache friendly way. In this format, each vertex stores the index (IA) into an array (JA), which stores in consecutive indices, the ids of adjacent vertices for each vertex in the graph. Thus the edges which vertex i is incident, can be identified as $JA[IA[i]]$ to $JA[IA[i+1]]$. The second is edge-centric and is best exemplified by the *coordinate list (COO)* format. In this format, each edge is represented as a pair of vertex ids (u, v), where u and v are the endpoints of the edge. This means that the topology of the graph is expressed as a list of endpoint pairs. Each of these formats is illustrated in Figure 1.

A. WebGraph Compression

It has been shown in [9] that web graphs and even online social network graphs exhibit two particular properties which can be exploited to achieve compact representations. Namely, *similarity* and *locality*. Similarity states that given an appropriate ordering of the vertices of the graph, vertices which have ids that are close together in number, will have similar adjacency lists. Locality means that given the same ordering, the vertex ids in a given vertex' adjacency list should be close to the id of the vertex itself. Leveraging these two properties, the authors of [9] developed the WebGraph Framework for compression of web graphs.

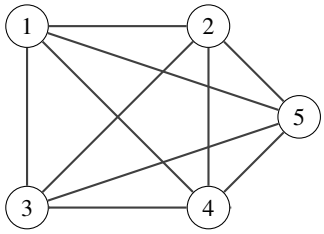
In the case of web graphs, a lexicographic ordering of the vertices based on their corresponding URL produces an ordering that satisfies the similarity and locality properties. This is because most links contained in a web page are of navigational nature, i.e., they point to other pages within the same domain. This means that most links will exhibit locality, since the ordering of the pages lexicographically means that pages within the same domain will appear close in the ordering. Furthermore, since most navigational link structures are shared among several pages within a domain, the links associated with a group of pages sharing navigational structure will exhibit similarity.

To exploit these characteristics for compression of web graphs, the authors of [9], employ the following three techniques. First, they use reference compression to leverage the similarity of adjacency information. This means that instead of representing the adjacency list of a particular vertex u directly, they represent it as a reference to another similar vertex v. After reference compression of $\Gamma(u)$, an intervalization technique is applied. Any vertex ids in $\Gamma(u)$ which did not appear in $\Gamma(v)$ are represented as a set of intervals and a list of residual ids. An *interval* is a consecutive sequence of vertex ids, which are common assuming locality of adjacency lists. Each interval is encoded as a start id and a length, so the sequence of vertex ids $3, 4, 5, 6, 7, 8$, would be encoded as $(3, 6)$. Any vertex ids of $\Gamma(u)$ which did not appear in $\Gamma(v)$ or any interval is encoded in the residual list. The start ids of each interval is differentially encoded, i.e., stored as the difference from the previous start id. The same is true of the residual list. A more detailed explanation of the WebGraph format can be found in [9]

The above discussion leverages the ability to lexicographically sort the vertices of web graphs based on their URLs to expose locality and similarity. However, in the case of online social graphs, lexicographically sorting the vertices does not guarantee the ability to expose locality and similarity. Thus, it is necessary to develop ordering schemes for online social networks which are conducive to this type of compression. Many orderings have been developed along these lines including BFS [5], Grey [6], Shingle [7], SlashBurn [10], and LLP [11].

IV. Problem Statement

As graph analysis techniques continue to evolve, the desire to perform this analysis in a dynamic context is ever increasing. In this work we investigate data structures for dynamic sparse graphs, with special attention being paid to memory requirements and access times. Specifically, given a graph $G = (V, E)$, we want to represent G using as little memory as possible and at the same time, achieve access times which are as low as possible. Due to the nature of sparse graphs, $|V| \ll |E|$, edge updates are significantly more



CSR	
IA	JA
0	2, 3, 4, 5
4	1, 3, 4, 5
8	1, 2, 4, 5
12	1, 2, 3, 5
16	1, 2, 3, 4
20	

COO	
(1, 2)	(3, 4)
(1, 3)	(3, 5)
(1, 4)	(4, 1)
(1, 5)	(4, 2)
(2, 1)	(4, 3)
(2, 3)	(4, 5)
(2, 4)	(5, 1)
(2, 5)	(5, 2)
(3, 1)	(5, 3)
(3, 2)	(5, 4)

Fig. 1. Example graph (left) along with its CSR format (middle) and COO format (right).

TABLE I
STORAGE AND UPDATE COMPLEXITY.

Data structure	Storage	Edge Update
LL	$\Theta(\|V\| + 2\|E\|)$	$O(\Delta)$
BCSR	$O(\|V\| + 2\|E\|)$	$O(\Delta)$
DAA	$\Theta(\|V\| + \|E\|)$	$O(\Delta)$
DIAA	$O(\|V\| + \|E\|)$	$O(\Delta)$
DCAA	$O(\|V\| + \|E\|)$	$O(3\Delta)$

* BCSR also incurs a cost to fold LL into static CSR, which is $O(|E|)$ over all folds

likely than vertex updates. For this reason, edge updates will be the focus of this paper. However, an algorithm for vertex updates is presented and is included in the experimental results. Furthermore, in each of the data structures described in this work, an algorithm for edge removal can easily be deduced from the algorithm for edge insertion. Thus, due to lack of space, we will not go into detail on edge removal algorithms.

V. DYNAMIC GRAPH REPRESENTATIONS

We investigate five data structures for representing dynamic sparse graphs. In each approach, a different trade-off is made to improve memory requirements or access time. Asymptotically, each of the five approaches exhibits identical behavior, namely, $O(|V| + |E|)$ memory and $O(\Delta)$ time for edge update; however, in practice, memory access patterns and internal data structure representation can cause significant variations in absolute memory used or access time. The $O(\Delta)$ update complexity is a direct result of the requirement that $\forall u \in V$, $\Gamma(u)$ is stored in sorted order. Maintaining this invariant is typical of many applications, due to its requirement for many graph operations, i.e., finding common neighbors of a vertex (set intersection). A sixth approach, based on the packed-memory array data structure described in [12] and applied to dynamic graphs in [13] was also implemented and experimented. However, due to the complex maintenance procedure required for edge updates, its runtime was prohibitively high and will not be discussed further.

A. Linked-List (LL)

The first data structure, referred to as *Linked-List (LL)*, is based on the standard singly-linked list representation of a sparse graph adjacency list. In LL, each vertex is an object which has a linked-lists of nodes. Each node in the linked-list contains the vertex of a single adjacent vertex. The memory required for this data structures is $\Theta(|V| + 2|E|)$ and the cost to update an edge is Δ, since the adjacency list needs to be maintained in sorted order [14].

B. Batch Compressed Sparse Row (BCSR)

There are two main drawbacks with LL. The first is that inserting into a linked-list while maintaining some ordering gets more expensive as the size of the data structure grows. Second, due to the pointer-based nature of linked-lists, the memory access patterns of linked-lists tend not to be cache friendly, making access to consecutive nodes of a linked-list more costly than access to consecutive elements of an array.

In BCSR the graph is represented as a combination of two data structures: a static CSR and a LL. Each time that a new edge is added, it is added directly into the LL. Again, the linked-list for vertex u and v are updated for edge (u, v). When the LL has grown significantly large, so that it is no longer memory or access efficient, it is folded into the static CSR. This means that the static CSR is appropriately resized to hold all edges from the graph. Then the static CSR is rebuilt to include the edges which it contained previously, as well as the edges from the LL. As edges are included in the static CSR, they are removed from the LL. Upon completion of a fold routine, the static CSR contains all edges currently in the graph and the LL is empty.

This type of batch processing approach has two benefits. First, by periodically reducing the size of the linked-list, the insertion time will be reduced. Second, the majority of the graph is stored in a data structure which has better properties for data access than the linked-list. Since BCSR is built upon both a static CSR representation as well as a LL, the memory requirement will be $\Omega(|V|+|E|)$ and $O(|V|+2|E|)$, depending on the interval at which the LL is folded into the static CSR. The trade-off in this type of approach is determining the appropriate time to fold the LL into the static CSR. Many policies can be designed to govern this decision, including, but not limited to, simple schemes such as folding after a constant number of updates have occurred, to more complex schemes based on timing data structure operations and folding when it becomes more expensive to operate on the unfolded data structure than to fold it and then operate.

Like the memory requirements for BCSR, the cost of updating an edge will depend on the interval at which the data structure is folded and will be $\Omega(1)$ and $O(\Delta)$. However, along with the cost of edge updates comes the cost of incorporating

Vertex	Degree	Adjacency array
...	...	...
13	8	1, 2, 3, 8, 9, 10, 11, 14
14	4	13, 15, 17, 42
15	5	14, 32, 33, 34, 35
...	...	...
32	8	2, 3, 4, 5, 10, 11, 12, 15
33	6	1, 2, 3, 5, 15, 23
34	5	1, 2, 3, 5, 6, 8
...	...	...

Fig. 2. Adjacency array information for a subset of vertices from an example graph.

Vertex	Degree	Intervals	Residuals
...	...	...	...
13	8	[1, 3], [8, 4]	14,
14	4		13, 15, 17, 42
15	5	[32, 4]	14
...	...	...	...
32	8	[2, 4], [10, 3]	15
33	6	[1, 3]	5, 15, 23
34	6	[1, 3], [5, 2]	8
...	...	...	...

Fig. 3. Intervalized adjacency array for example graph from Figure 2.

Vertex	Degree	Intervals	Residuals
13	9	[1, 3], [8, 4], **[14, 2]**	
15	6	[32, 4], **[13, 2]**	

Fig. 4. Adding edge (13,15), which creates a new interval in each vertex object, shown in bold.

the LL into the static CSR. This requires iterating all edges currently in the graph. As long as the size of the static CSR is expanded by some constant proportion, it can be shown that the amortized cost of folding the LL into the static CSR carries a cost of $O(|V| + |E|)$ over all folds [14].

C. Dynamic Adjacency Array (DAA)

Implementing a data structure based on a linked-list means that there will always be a 2x memory overhead, since each linked-list node must store a value as well as a pointer to the next node. In the third implementation, called *Dynamic Adjacency Array (DAA)*, this memory overhead is eliminated by assigning to each vertex a dynamically allocated array in which to stores the ids of adjacent vertices. The size of this dynamically allocated array is equal to exactly the current degree of the vertex. In order to iterate or search the adjacency array of a particular vertex it is necessary to know its current degree. Thus, this value is also stored and maintained across updates as part of the vertex object, along with the adjacency array. The storage of degree along with adjacency arrays in each vertex object make the memory requirement of this data structure $\Theta(|V| + |E|)$.

When a new edge (u, v) is added to the graph, the adjacency arrays for vertices u and v are updated as before. To do this, each of their adjacency arrays must be resized to accommodate the new adjacent vertex id. After resizing the array, the index where the new id belongs is identified and all ids following the index are shifted to the next index, at which point, the new vertex id is inserted. Finally, the degree of the vertex is incremented by one to reflect the addition of the new edge. Each edge addition will require the dynamic adjacency array to be reallocated to a larger size array, which in the worst case, requires copying the entire array from the old memory location to the new. For that reason, similar to the previous data structures, the cost of edge updates in DAA is $O(\Delta)$.

D. Dynamic Intervalized Adjacency Array (DIAA)

DAA addresses many of the issues of the previous data structures. However, at best, that data structure will require no less memory than storing a static snapshot of the entire graph. As the size of graphs continues to rise, it is often desirable to use a structure whose memory requirements are less than $\Theta(|V| + |E|)$ in practice. It is for exactly this case that we explore two final data structures, the first of which we refer to as *Dynamic Intervalized Adjacency Array (DIAA)*. This data structure is a subtle variation of DAA, which provides a significant reduction in memory requirements for certain classes of graphs.

The DIAA data structure is similar to DAA in that each vertex is stored as an object with a degree and an adjacency list representation. However, whereas in DAA, the adjacency list was represented explicitly using an adjacency array, in DIAA it is represented using an array of intervals and residual ids, see Section III-A. This simple optimization has the potential to reduce the memory requirements and improve the performance for a dynamic graph. If a graph exhibits locality, intervals will exist in vertex adjacency lists, and thus, memory requirements will be reduced.

The DIAA format can be specified as follows:

$$\delta\Big[i\big[E_1L_1\cdots E_iL_i\big]_{i>0}\big[R_1\cdots R_k\big]_{k>0}\Big]_{\delta>0}$$

where subscripted brackets denote the condition required for the enclosed portion of the format to be present, i is the number of intervals, and $k = \delta - \sum_{j=1}^{i} L_j$, is the number of residuals.

Furthermore, by intervalizing the adjacency arrays, update and query operations can also be made more efficient. For example, when an edge is inserted into a DAA, the correct index for the new vertex id is identified and all elements subsequent to the index are shifted by one array index to make room for the new id. Contrast this with insertion into a DIAA. In a DIAA data structure the insertion of a new vertex id into a vertex object will have one of the following effects: 1) create a new interval, 2) extend an existing interval, or 3) become a residual. Examples of creating a new interval and extending an interval can be seen in Figures 4 and 5. There are three types of interval extensions, each of which is illustrated in Figure 5. The first type, *int-new*, happens when a new vertex id is added which is equal to one more or less then the highest or lowest id in an interval. In this case, the interval cannot be further extended by any of the residuals, which is exactly the second type, called *int-new-res*. The final type, called *int-int*, occurs when the new vertex id is exactly one less than the lowest id of one interval and exactly one more than the highest id of another interval.

Since an intervalized adjacency array can require at most $O(1)$ more space than its explicit counterpart, for storing the

Vertex	Degree	Intervals	Residuals	Type
15	6	**[31, 5]**	14	int-new
33	7	**[1, 5]**	15, 23	int-new-res
34	7	**[1, 6]**	8	int-int

Fig. 5. Adding three edges, (4, 33), (4, 34), and (15, 31), to illustrate the three different interval extensions possible, changes denoted by bold text.

Vertex	Degree	Intervals	Residuals
...	...	...	...
13	8	[1, 3], [8, 4]	2
14	4		1, 4, 4, 50
15	5	[32, 4]	1
...	...	...	...
32	8	[2, 4], [10, 3]	33
33	6	[1, 3]	55, 10, 8
34	6	[1, 3], [5, 2]	43
...	...	...	...

Fig. 6. Compressed adjacency array, before integer coding, for example graph from Figure 2.

number of intervals in its vertex object, memory requirements for DIAA are expressed as $O(|V|+|E|)$. This also means that scanning and shifting of an intervalized adjacency array, which require at most the same number of memory operations as the explicit counterpart, incur a cost of $O(\Delta)$.

E. Dynamic Compressed Adjacency Array (DCAA)

It is possible that the dynamic graph being represented does not exhibit the locality required to exploit an intervalized representation. It is also possible that the memory reduction achieved by intervalization alone is insufficient. In these cases, intervalization may not provide a satisfactory reduction in memory requirements, and an additional layer of compression applied on top of the intervalized representation may be attractive. To this end a final data structure is investigated. Referred to as *Dynamic Compressed Adjacency Array (DCAA)*, this data structure leverages ideas from the WebGraph framework described in Section III-A to achieve an even more compact representation than DIAA. While DCAA benefits from the locality required for a successful memory reduction in DIAA, it also has potential to reduce memory requirements for all classes of dynamic graphs, as it is based on differential encoding and a compact representation of integer values.

In this representation, each vertex object stores the ids of adjacent vertices in a modified WebGraph compressed array. When a new edge (u, v) is added to a DCAA, the initial behavior is much like that of the DIAA, i.e., two adjacency arrays are updated to reflect the addition of the new edge. However, in the case of DCAA, simply resizing and insertion is insufficient. Since the DCAA stores each adjacency array in compressed intervalized form, and the modified WebGraph compression does not allow for modification in compressed form, it is necessary to first decompress the intervalized adjacency array being updated. At this point, the new vertex id can be inserted following exactly the same procedure as DIAA. After insertion, the intervalized adjacency array is compressed again and the insertion is complete. The memory accesses required for the decompression, insertion, compression cycle is at worst, three scans over the entire adjacency array (case with no intervals), making the cost for edge update in DCAA to be $O(3\Delta)$.

TABLE II
DATASETS USED FOR EXPERIMENTAL EVALUATION.

Dataset	$\|V\|$	$\|E\|$	Type
in-2004	1382908	27182946	web
eu-2005	862664	32276936	web
lj-2008	5363260	39511571	social
com-orkut	3072441	234370166	social

F. Vertex Updates

The previous discussion addressed only the circumstances related to the dynamic update of the edges of a graph. In many cases, the range of vertex ids will not be known a priori. In these cases, the graph data structure must also support vertex updates. For LL, BCSR, and DAA, these operations are straight-forward to implement, since the same ideas which were used to support edge updates can easily be extended to vertex updates. The same is true of DIAA and DCAA, with one single, but important caveat. The caveat being that the memory and access efficiency of the DIAA and DCAA data structures depend highly on the ordering of the vertices. Under dynamic vertex updates, maintaining the optimal vertex ordering for these two data structures is a hard problem.

Since typical web and social graphs have significantly more edges than they do vertices, $|V| \ll |E|$, the number of edge updates will usually be at least an order of magnitude higher than the number of vertex updates. Therefore, it may be computationally feasible to periodically reorder the graph, assuming the existence of an efficient graph reordering technique designed to expose intervalization. The BFS reordering of [5] and the Grey code reordering of [6] both take linear time on the number of vertex and edges, and thus meet the requirements for the vertex update scheme. A slightly more computationally expensive approach, which has been shown to result in a more favorable vertex ordering for WebGraph based schemes is the Layered Label Propagation reordering of [11]. This method however, requires several iterations of a linear time reordering algorithm to achieve good results, thus should be reserved for the scenario when the maximum amount of compression is needed, regardless of the cost. Between reorderings, vertices can be assigned ids based on their chronological appearance. This introduces a trade-off between how often to reorder and the effectiveness of the intervalization mechanisms of DIAA and DCAA. Since vertex updates are infrequent compared with edge updates, a conservative approach with a smaller interval between reorderings can be employed to ensure intervalization effectiveness.

VI. Experimental Evaluation

A. Datasets

We evaluated our algorithms using five real world datasets obtained from the Laboratory for Web Algorithmics, http://law.di.unimi.it. The first two datasets are examples of web graphs, while the latter two are online social graphs. Their characteristics are shown in Table II. As can be seen, in all

datasets, the number of edges is greater than the number of vertices by at least an order of magnitude.

B. Implementation

All data structures and algorithms were implemented in C. The experiments were run with a single thread on an Intel (R) Xeon (C) 2.80GHz CPU. The test applications were compiled on Ubuntu 10.04 with GCC version 4.5.2. Timing was performed using the *gettimeofday()* POSIX.1-2001 function.

Through experimental evaluation, it was determined that the *malloc()* implementation shipped with the standard C library on Ubuntu 10.04 was unsuitable for the dynamic nature of many of the data structures described in this work. This is especially true for DAA, DIAA, and DCAA, whose memory cannot be bulk allocated like that which can be done when allocating nodes for a linked-list or the single allocation required to resize a static CSR structure. For this reason, we implemented our own malloc() library that is specifically designed for the way that memory is allocated for DAA, DIAA, and DCAA. Recall that the graphs which are being focused on here, namely web and online social graphs, tend to exhibit scale-free behavior. This means that the majority of vertices in the graph have a small degree. Since each adjacency list in DAA, DIAA, and DCAA requires a separate allocation, malloc() from the standard C library, incurs a great deal of bookkeeping overhead. This overhead was eliminated in the alternative malloc implementation by using the well known fixed-size block allocation technique instead of the linked-list management technique of many malloc implementations. By eliminating this memory overhead, the amount of virtual address consumed by an application utilizing dynamic data structures like those described here can be greatly reduced.

With regards to BCSR, we employ a policy for folding such that the LL gets incorporated into the static CSR only when a query algorithm is to be executed on the data structure. The advantage of this is that due to the good access characteristics of the static CSR, algorithms executed on the data structure after folding will be very efficient. The drawback of this approach is that the LL can grow quite large between algorithm executions. However, if algorithms are executed with sufficiently small intervals between, then the memory requirements of this approach will not grow undesirably high.

Finally, for the procedure of vertex reodering, we chose two strategies based on the class of graph under consideration. For web graphs, the lexicographic ordering of vertex URLs was used to impose an ordering on the vertices. This was shown in [15] to lead to a very compression friendly ordering, and is extremely efficient in implementation. For social graphs, an ordering based on the BFS ordering of [5] was used.

C. Performance Assessment Metrics

For the experiments, each dataset was processed independently. Edges were chosen at random from each edge set E to be added to the data structure being tested. Each edge was added exactly one time and edges were added in the same order across all data structures.

We measured the performance of the various approaches along three dimensions: (i) memory usage, (ii) update time, and (iii) access time. Memory usage was reported using two different metrics. The first was the amount of memory which each data structure occupied at any given time. This measure closely resembles the theoretical memory requirements, as it reports only the exact number of bytes needed to store each data structure. The second measure is virtual address space usage. This measure reflects the amount of memory obtained from the OS by the benchmark application to persist and operate on each data structure and is directly affected by the OS and malloc implementation's treatment of allocated memory.

The amount of time to update the graph data structures was measured as a function of the number of edges in the graph. The update execution time is reported as the aggregate number of milliseconds spent on edge addition since the last query operation.

D. Benchmark Applications

To measure execution times for graph access, the following policy was used. The graph structures were queried after every $|V|/\alpha$ edge additions. For the results presented in Section VII, α was set to be 200. Each $|V|/\alpha$ interval will be furthermore referred to as a *snapshot*. Three kernel benchmarks were used to measure the graph access time. Each of the kernels was chosen to stress different graph access patterns. We believe that these three benchmarks are representative sample of graph kernels, which can be used as building blocks for more complex analysis.

The first kernel is the breadth-first traversal (BFT). BFT is an edge traversal which uses a queue to maintain the order in which to visit vertices. At each iteration, the edges of the current vertex are expanded and any vertices which have not already been enqueued, are added to the queue. The next iteration commences by popping the next vertex from the queue. If the queue is empty, but not all vertices have been visited, a random vertex is chosen which has not been visited. This is repeated until all vertices have been visited.

The second kernel is a neighbor query. The purpose of this query is to identify the set of vertices which two vertices have in common and is implemented as a set intersection in practice. Typically, a set intersection requires iterating all the elements of each of the sets under consideration and testing for equality. If the sets are sorted, this can be done in a single pass over the two sets and only requires as many comparisons as there are elements is the smaller of the two sets. There are more complex algorithms for intersection of arrays, as opposed to linked-lists, which can be made extremely fast in practice, such as [16]. This operation can be further optimized for interval aware data structures like DIAA and DCAA, where instead of iterating over all adjacent edges of the two query vertices, processing proceeds as follows. Given two intervalized arrays, find the smallest element in each of the lists. This element can exist within the bounds of the first unprocessed interval or as the first unprocessed residual. Then, given the smallest element of each array, a comparison is made exactly as it would be for a standard set intersection. However, instead of iterating through each array comparing every element, when the elements of the two arrays are not equal, it is possible to skip entire intervals while only performing a comparison of their start id and end

TABLE III
NUMBER OF INTERVALIZED EDGES IN EACH GRAPH DATASET.

Dataset	Intervalized Edges	% of Total Edges
in-2004	21386085	78.67
eu-2005	22623251	70.09
lj-2008	11288594	14.29
com-orkut	11083262	4.73

TABLE IV
MEAN LENGTH (STANDARD DEVIATION) OF LINKED-LIST BEING UPDATED FOR EACH DATASET UNDER LL AND BCSR DATA STRUCTURES.

Dataset	LL		BCSR	
in-2004	555.76	(1527.16)	2.78	(7.81)
eu-2005	1288.62	(4806.19)	6.44	(24.16)
lj-2008	53.33	(102.99)	0.23	(1.16)
com-orkut	194.67	(900.50)	0.97	(4.61)

id, thus potentially saving a significant number of comparisons. Furthermore, two intervals can be intersected directly without iterating either of their ranges in its entirety.

The final kernel is the adjacency query. This is a simple check whether or not an adjacency list contains a specified vertex id. The standard approach is a linear search for linked-list and a binary search for arrays. However, like the neighbor query, adjacency queries can also be optimized for interval aware data structures. The idea is the same, instead of performing a binary search over every vertex id in an adjacency array, a modified binary search should be employed which makes a distinction between intervals and residuals and has the ability to treat intervals as singular elements.

In our experiments, the set of query vertices was kept consistent within each kernel benchmark. For example, in the BFT kernel, the same set of vertices was uses as the root of the BFT regardless of the data structure being tested. The same is true of the neighbor and adjacency queries.

VII. RESULTS

A. Memory Requirements

Figures 7 and 8 show the memory requirements for each of the five dynamic graph data structures per dataset. From Figure 7, we see that the memory required by each data structure, follows the same pattern as the theoretical bounds for memory requirements for each data structure. As expected, intervalization and integer coding reduces the memory required in all cases. However, the scale of the reduction is much higher in the case of web graphs, where adjacency lists are typically well intervalized, as shown in Table III.

Also interesting is the virtual memory required by each of the data structures, as can be seen in Figure 8. In the case of virtual memory, the footprint of each data structure follows the same pattern as requested memory, except for BCSR, which has the highest virtual memory requirements by nearly 20%. This can be attributed to the extra memory required to fold the linked-list portion of the graph into the static CSR, which is an important consideration if memory consumption during execution must be limited.

B. Update Time

The second measure used to evaluate the data structures was update time. Figure 9 shows the execution time for the addition of edges for each snapshot. Note that the execution time for a snapshot includes only those edges which were added during a particular snapshot. As expected BCSR has the lowest edge addition time, when considered without the cost of folding. This is based on its reliance on linked-lists, which are very efficient for short lists. Since, BCSR offloads edges from its linked-lists into a static CSR occasionally, the linked-lists which it requires tend to be of shorter length, which can be seen in Table IV. However, when folding is considered, BCSR w/ fold, DIAA actually surpasses BCSR for sufficiently large and intervalized graphs, as would be expected based on the discussion in Section V-D.

One might expect that DCAA would have the slowest edge addition time for all datasets, since the overhead of decompression and compression will have a negative impact on addition time. However, as Figure 9 shows, DCAA is actually faster than LL at later snapshots. As mentioned in Section V-D and supported by Figure 9 (DAA vs. DIAA), adding an edge into an intervalized adjacency array is at least as efficient as inserting into an explicit adjacency array. Since ordered insertion into a linked-list is slower than insertion into an array (LL vs. DAA) and the web graphs are highly intervalized, the efficiency introduced by intervalization allows the DCAA data structure to support edge additions at a higher rate than LL. The same is true for the social graphs, but to a lesser degree. However, even for intervalized datasets, DCAA still requires more time for edge addition than the rest of the data structures.

C. Performance of Benchmark Applications

The results for the three kernel benchmarks discussed in Section VI-C are shown in Figures 10 to 12, respectively. Figure 10 shows the results of the BFT kernel benchmark, where it can be seen that BCSR exhibits the fastest overall execution time, with DAA and DIAA close behind. This is exactly as we would expect, since the representation of the graph during operation in the BCSR data structure is in static CSR format. Even with folding time considered, BCSR is still fastest, which is a consequence of the nature of the query, namely, a traversal of the entire graph. Since graph traversal in BCSR has the same complexity as edge folding, but a less friendly memory access pattern, we would expect that the graph traversal cost will outweigh the cost of edge folding.

The DAA data structure stores the graph in conceptually the same way as BCSR, but because each of the adjacency lists in DAA is allocated dynamically, the data locality that can be exploited in BCSR is not present to the same degree in DAA. The same is true of DIAA, with the addition of a slight overhead due to the bookkeeping of the intervalized adjacency lists and DCAA with a further overhead of integer coding. DCAA executes between 3.5 and 4.5 times slower than BCSR, which can be attributed to the overhead of decompression of the adjacency lists before operation. Finally, LL consistently reports the slowest execution times for BFT.

In the case of the neighbors query, a set intersection is performed between two randomly chosen vertices. As Figure 11

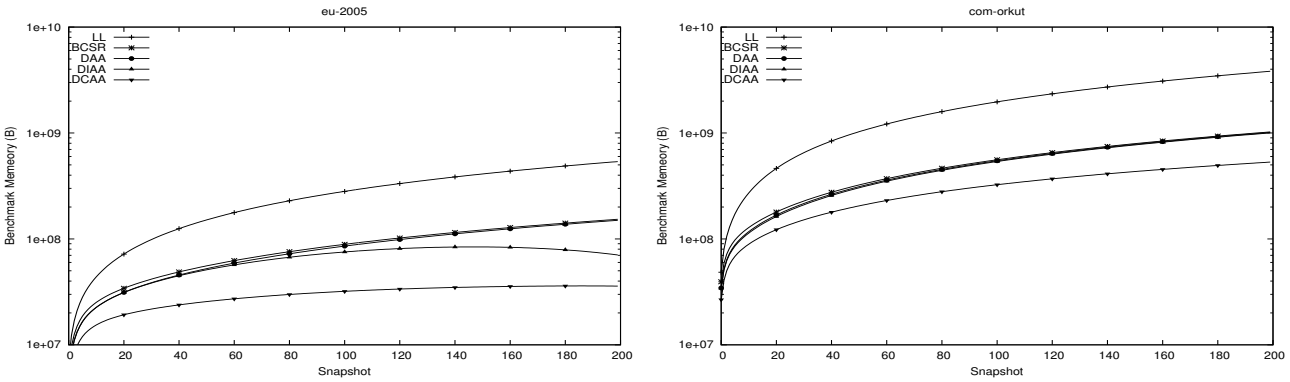


Fig. 7. Required memory for each data structure.

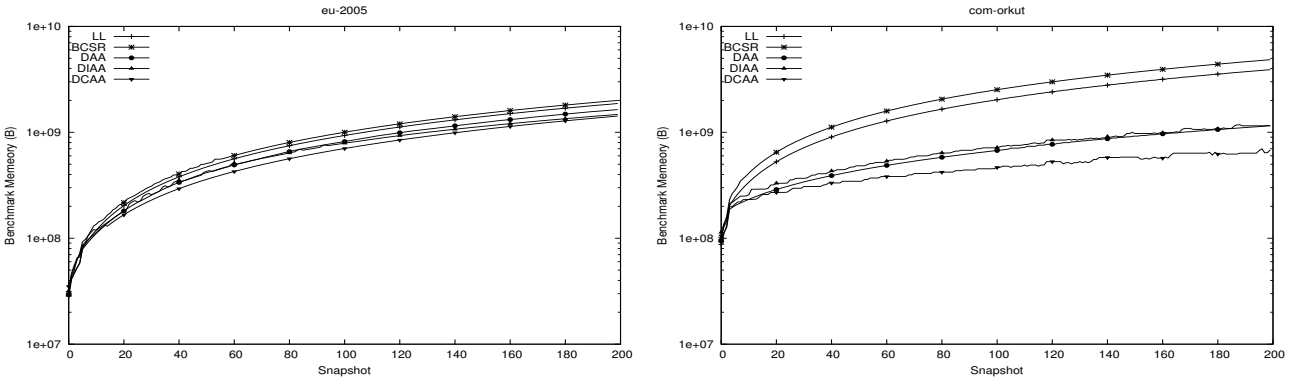


Fig. 8. Virtual memory usage during benchmark application.

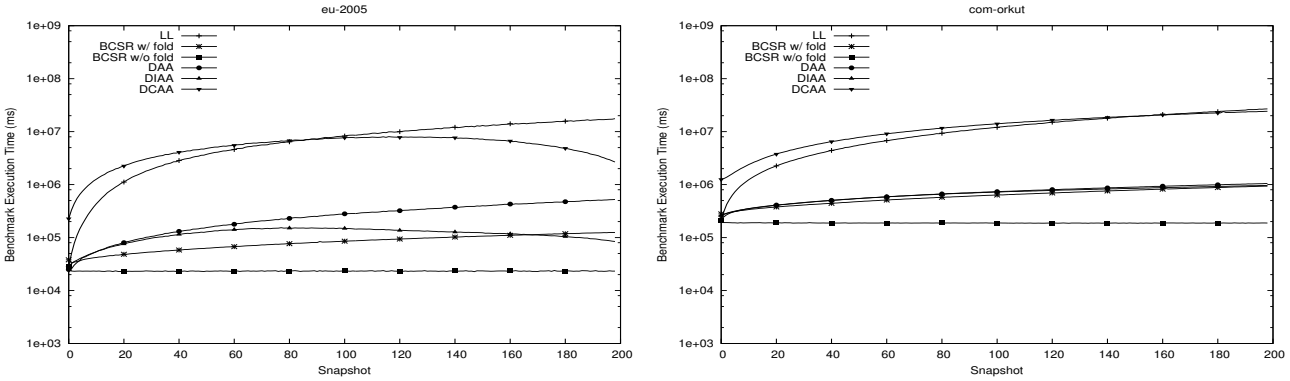


Fig. 9. Edge addition times.

shows, the results of performing a set intersection are similar for LL, BCSR, and DAA, but quite different when compared with the results for BFT for the other two data structures. As alluded to above, this is due to optimizations which can be exploited by the interval aware data structures. In any of the first three data structures, a basic set intersection requires iterating all the edges of at least one of the participating vertices. In the case that the vertex degrees of the two vertices being intersected are different, only the adjacency list of the smaller degree vertex need be fully iterated. However, in the final two data structures, namely DIAA and DCAA, adjacency lists are stored as intervals and residuals. In these cases, rather than iterating entire adjacency lists, it is only necessary to iterate the intervals and residuals. If the adjacency lists are well intervalized, then the number of intervals and residuals will be much less than the degree of the vertex, meaning a set intersection can be performed with many fewer comparisons.

For web graphs, which are well intervalized, this results in in DIAA having the fastest execution time for neighbor queries. Further, the access time of DCAA also comes very close to surpassing that of the non-intervalized data structures. This illustrates an important characteristic of the two classes of data structures. Namely, that those data structures which are not interval aware will have access times which are non-decreasing as a function of the number of edges in the graph, while interval aware data structures will exhibit decreasing access times as the number of edges grows and the intervals become more dense, assuming an interval friendly ordering of

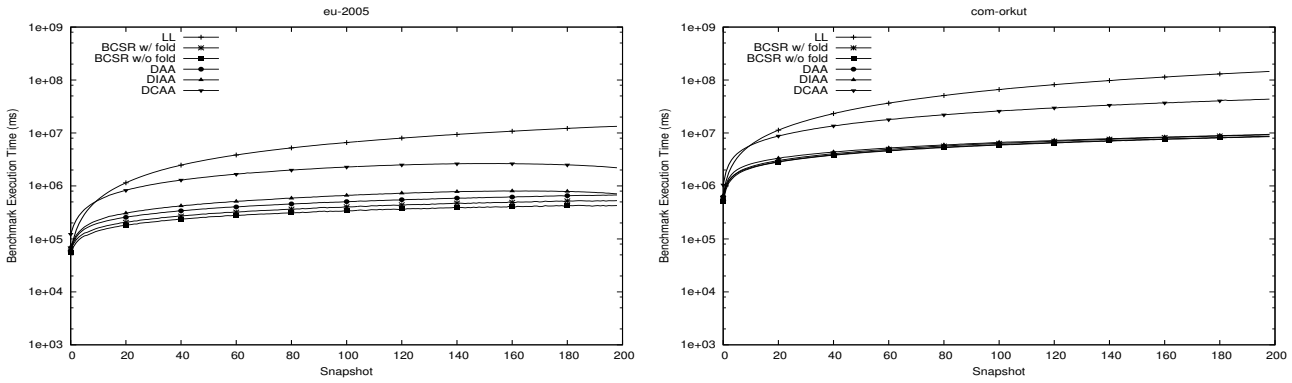


Fig. 10. Breadth-first traversal.

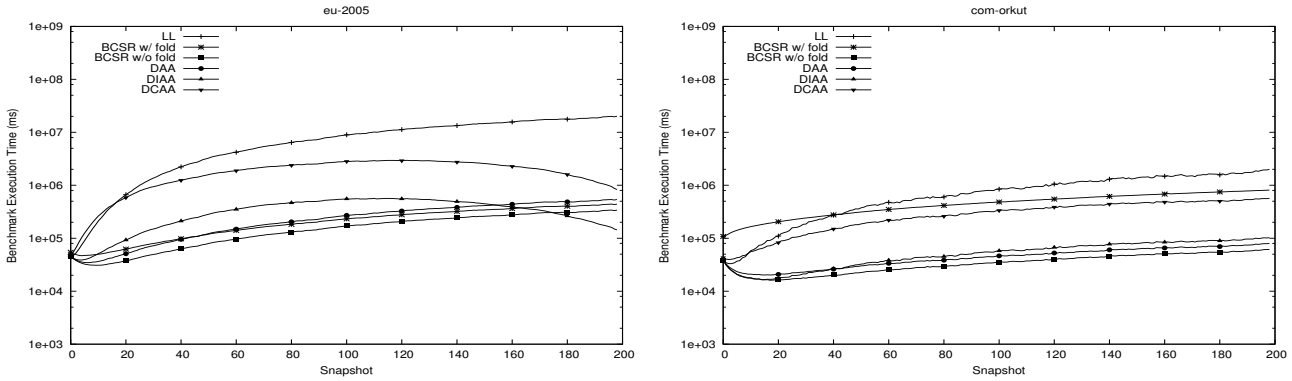


Fig. 11. Neighbor query.

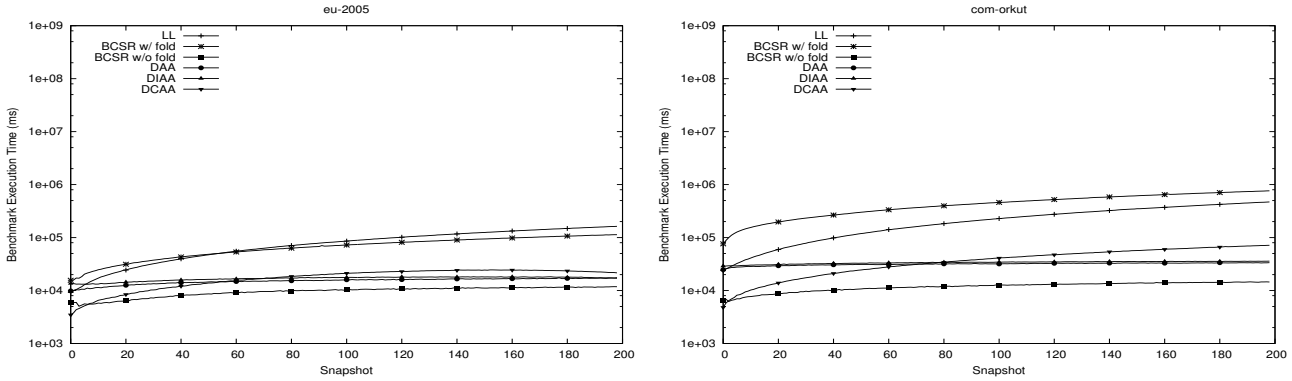


Fig. 12. Adjacency query.

the vertices.

The final query type tested was the adjacency query. At first glance, an adjacency query is similar to a neighbors query, since both require the scanning of particular adjacency lists. However, just as neighbors queries can be optimized to leverage an intervalized data structure, so too can adjacency queries. This is especially true in the case of the DCAA data structure. Instead of decompressing the entire array then performing a binary search, which costs $O(n + \log n)$, each element of the array can be checked as it is decompressed and the search can return as soon as the desired element is found, without decompressing the rest of the array. The cost of the optimized search is $O(n)$, which is slightly better than decompressing and performing a binary search. As Figure 12 shows, this type of optimizations means that the DCAA data structure is very competitive in terms of adjacency query execution time. It is approximately 3x slower at worst, and less than 2x slower in most cases.

D. Selection of α for BCSR

There is an inherent trade-off to be made if an α based policy is chosen for edge folding in BCSR. Fold too often and the cost of the edge folding operation outweighs the benefits it gives in terms of graph update and access times. Figure 13 shows that there is a sweet-spot in terms of α. For the web graph dataset, eu-2005, the best α value in terms of total benchmark suite execution time is 50. For the social graph dataset, com-orkut, this value is 10. The most likely

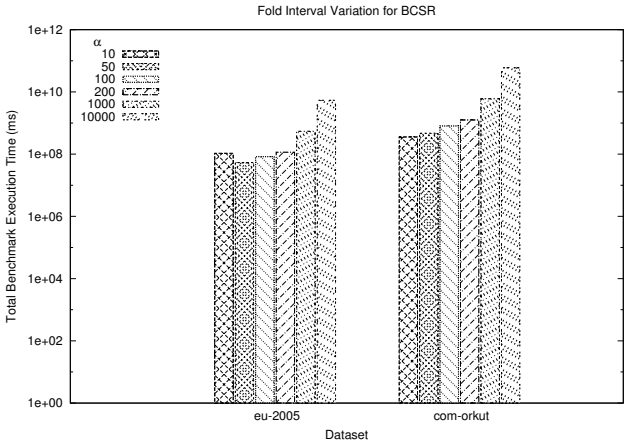


Fig. 13. Benchmark suite execution time under variations of α.

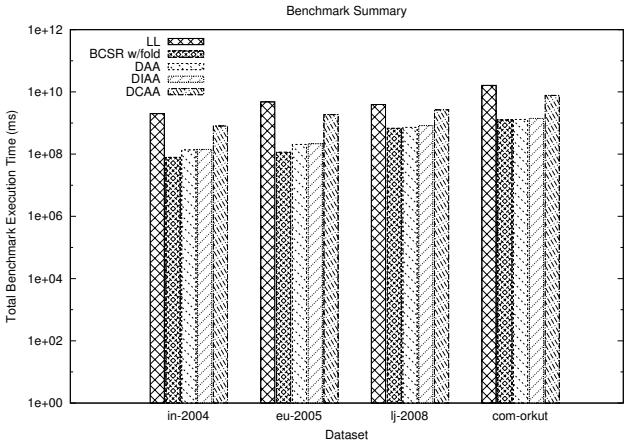


Fig. 14. Execution time required for benchmark suite for each data structure.

explanation for the effectiveness of small α for social graphs is due to their relatively small linked-list lengths, as shown in Table IV. Since the linked-lists are short, relative to those of the web graphs, the advantages of frequent folding are reduced, since one of the main goals of edge folding is to reduce the lengths of the linked lists.

E. Results on All Graphs

In our experiments, eu-2005 and com-orkut were representative of web graphs and social graphs respectively. For this reason and due to space limitations, detailed plots for the remaining three datasets have been omitted. However, Figure 14 shows a summary of benchmark suite execution time for each dataset. This figure supports conclusions drawn from the previously presented results in all cases except one, namely that BCSR is inferior to DIAA in terms of access time and query time. Figure 14 shows that the total execution time for the benchmark suite is smaller for BCSR than DAA. This is due to the fact that Figure 14 shows the total execution time for the benchmark suite and does not illustrate the following fact in the context of web graphs. In early snapshots, DIAA is slower, due to bookkeeping overhead, but in later snapshots, the intervalized structure of the web graphs, allows the update and access time to decrease in DIAA versus, a non-decreasing update and access time for BCSR, recall Figures 9 to 12.

VIII. Conclusions

We presented five data structures for maintaining dynamically updated graphs. Three of the data structures are well known and treated as base line approaches. One of the data structures, DIAA, is optimized for memory usage and update and access time for classes of graphs which exhibit locality. The last data structure, DCAA, is an extension of DIAA, optimized entirely for reduced memory requirements. Each data structure was evaluated theoretically and experimentally using four real world datasets. The experimental analysis shows DIAA is superior for web graphs and nearly as good as the others for social graphs. Further, our results show that, in all cases, DCAA is the most memory efficient data structure, at the cost of a modest increase in update and access time.

Acknowledgment

This work was supported in part by NSF (IIS-0905220, OCI-1048018, and IOS-0820730) and by the DOE grant USDOE/DE-SC0005013 and the Digital Technology Center at the University of Minnesota. Access to research and computing facilities was provided by the Digital Technology Center and the Minnesota Supercomputing Institute.

References

[1] R. Kumar, J. Novak, and A. Tomkins, "Structure and Evolution of Online Social Networks," in *Link Mining: Models, Algorithms, and Applications*. Springer, 2010, pp. 337–357.

[2] M. Thorup, "Near-optimal fully-dynamic graph connectivity," in *Proceedings of the thirty-second annual ACM symposium on Theory of computing*. ACM, 2000, pp. 343–350.

[3] L. Roditty and U. Zwick, "On dynamic shortest paths problems," *Algorithmica*, vol. 61, no. 2, pp. 389–401, 2011.

[4] V. Stix, "Finding all maximal cliques in dynamic graphs," *Comput. Optim. Appl.*, vol. 27, no. 2, pp. 173–186, Feb. 2004.

[5] A. Apostolico and G. Drovandi, "Graph compression by bfs," *Algorithms*, vol. 2, no. 3, pp. 1031–1044, 2009.

[6] P. Boldi, M. Santini, and S. Vigna, "Permuting web graphs," in *Algorithms and Models for the Web-Graph*. Springer, 2009, pp. 116–126.

[7] F. Chierichetti, R. Kumar, S. Lattanzi, M. Mitzenmacher, A. Panconesi, and P. Raghavan, "On compressing social networks," in *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2009, pp. 219–228.

[8] C. Karande, K. Chellapilla, and R. Andersen, "Speeding up algorithms on compressed web graphs," in *Proceedings of the Second ACM International Conference on Web Search and Data Mining*, ser. WSDM '09. New York, NY, USA: ACM, 2009, pp. 272–281.

[9] P. Boldi and S. Vigna, "The webgraph framework i: Compression techniques," in *Proceedings of the 13th international conference on World Wide Web*. ACM, 2004, pp. 595–602.

[10] U. Kang and C. Faloutsos, "Beyond 'caveman communities': Hubs and spokes for graph compression and mining," in *Proceedings of the 2011 IEEE 11th International Conference on Data Mining*, ser. ICDM '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 300–309.

[11] P. Boldi, M. Rosa, M. Santini, and S. Vigna, "Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks," in *Proceedings of the 20th international conference on World Wide Web*. ACM, 2011, pp. 587–596.

[12] M. A. Bender and H. Hu, "An adaptive packed-memory array," *ACM Transactions on Database Systems (TODS)*, vol. 32, no. 4, p. 26, 2007.

[13] G. Mali, P. Michail, and C. Zaroliagis, "A new dynamic graph data structure for large-scale transportation networks," eCompass, Tech. Rep., 2012.

[14] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein *et al.*, *Introduction to algorithms*. MIT press Cambridge, 2001, vol. 2.

[15] K. H. Randall, R. Stata, J. L. Wiener, and R. G. Wickremesinghe, "The link database: Fast access to graphs of the web," in *Proceedings of the Data Compression Conference*, ser. DCC '02. Washington, DC, USA: IEEE Computer Society, 2002, pp. 122–.

[16] R. Baeza-Yates, "A fast set intersection algorithm for sorted sequences," in *Combinatorial Pattern Matching*, ser. Lecture Notes in Computer Science, S. Sahinalp, S. Muthukrishnan, and U. Dogrusoz, Eds. Springer Berlin Heidelberg, 2004, vol. 3109, pp. 400–408. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-27801-6_30