Packed Compressed Sparse Row: A Dynamic Graph Representation

Brian Wheatman
Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology
wheatman@mit.edu

Helen Xu
Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology
hjxu@mit.edu

***Abstract*—Perhaps the most popular sparse graph storage format is Compressed Sparse Row (CSR). CSR excels at storing graphs compactly with minimal overhead, allowing for fast traversals, lookups, and basic graph computations such as PageRank. Since elements in CSR format are packed together, additions and deletions often require time linear in the size of the graph.**

We introduce a new dynamic sparse graph representation called *Packed Compressed Sparse Row* (PCSR), based on an array-based dynamic data structure called the Packed Memory Array. PCSR is similar to CSR, but leaves spaces between elements, allowing for asymptotically faster insertions and deletions in exchange for a constant factor slowdown in traversals and a constant factor increase in space overhead.

Our contributions are twofold. We describe PCSR and review the theoretical guarantees for update, insert, and search for PCSR. We also implemented PCSR as well as other basic graph storage formats and report our findings on a variety of benchmarks. PCSR supports inserts orders of magnitude faster than CSR and is only a factor of two slower on graph traversals. Our results suggest that PCSR is a lightweight dynamic graph representation that supports fast inserts and competitive searches.

***Index Terms*—sparse graphs, sparse matrices, storage formats, compressed sparse row, packed memory array, dynamic data structures.**

I. Introduction

Many real-world graphs such as the Facebook social network [UKBM11] are sparse in that the number of edges present in the graphs are much smaller than the possible number of edges. In practice, sparse matrices and graphs are often stored in compressed sparse row (CSR) format, which packs edges into an array and takes space proportional number of vertices and edges.

Sparse storage formats pay for these space savings with the cost of updates. CSR format supports fast queries such as membership or finding all neighbors of a vertex, but may require changing the entire data structure to add or remove an edge.

Social networks such as Facebook and Twitter are highly dynamic graphs since new users and connections are added constantly. Twitter averages about 500 million tweets a day [Say] and Facebook has about 41,000 posts (2.5Mb of data) per second [WKF+15].

Internet graphs are also highly dynamic: large Internet Service Providers (ISPs) field around 10^9 packets/router/hour [GM12]. Dynamic graphs have wide applications from recommendation systems to cellular networks and require efficient updates to graph storage formats.

Graph storage formats need to be able to support queries for edges, looping over all neighbors of a vertex, and looping over all edges. Common graph operations include breadth-first search (BFS) and PageRank [XG04], both of which involve a scan across all graph edges.

Sparse-matrix vector multiplication (SpMV) is a widely-used kernel in numerical and scientific computing and requires a scan over all nonzeros (edges). For example, iterative computations such as conjugate gradient are staples of numerical simulations and require repeated SpMV operations [Saa03]. One application of dynamic sparse graph representations is control-flow analysis, which involves successively extending a graph (adding nodes and edges) until it reaches a fixed point [Shi91].

Related Work

We present a dynamic data structure called packed compressed sparse row (PCSR) independent of any framework. PCSR is a graph representation rather than an dynamic analytics framework and can supplement existing graph analytics solutions. Existing dynamic graph analytics solutions such as GraphChi [KBG12], LLAMA [MMMS15], and STINGER [BBAB+09] [EMRB12] provide data structures for graph storage. These frameworks, however, often lack theoretical guarantees on performance. PCSR may be able to mitigate worst-case behavior in these graph frameworks as it has performance guarantees.

Sha *et al.* [SLHT17] introduced GPMA, a GPU-based dynamic graph storage format based on the packed memory array (PMA). GPMA handles concurrent inserts and is optimized for parallel batch updates. In this work, we focus on sequential random updates for CPUs rather than batched updates.

The most relevant related work is an in-place dynamic CSR-based data structure (DCSR) for GPUs due to King *et al.* [KGKM16]. DCSR is similar to PCSR in that it leaves extra space in each row. DCSR lacks theoretical guarantees on its runtime or space usage, however. Finally, it is only implemented for GPUs and not for CPUs.

Contributions

Our contributions are as follows:

- We describe a modification to compressed sparse row format, called packed compressed sparse row (PCSR)

978-1-5386-5989-2/18/$31.00 ©2018 IEEE

	Adjacency Matrix	Adjacency list (linked list)	Blocked Adjacency List	Compressed Sparse Row	**Packed CSR** (amortized)
Storage cost / scanning whole graph	$O(n^2/B)$	$O(n/B+m)$	$O((m+n)/B)$	$O((m+n)/B)$	$O((m+n)/B)$
Add edge	$O(1)$	$O(1)$	$O(1)$	$O((m+n)/B)$	$O(\lg^2(m+n)/B)$
Update or delete edge from vertex v	$O(1)$	$O(deg(v))$	$O(deg(v)/B)$	$O((m+n)/B)$	$O(\lg^2(m+n)/B)$
Add node	$O(n^2/B)$	$O(1)$	$O(1)$	$O(1)^*$	$O(\lg^2(m+n)/B)$
Finding all neighbors of a vertex v	$O(n/B)$	$O(deg(v))$	$O(deg(v)/B)$	$O(deg(v)/B)$	$O(deg(v)/B)$
Finding if w is a neighbor of v	$O(1)$	$O(deg(v))$	$O(deg(v)/B)$	$O(\frac{\lg(deg(v))}{\lg(B)})$	$O(\frac{\lg(deg(v))}{\lg(B)})$
Sparse matrix-vector multiplication	$O(n^2/B)$	$O(n/B+m+n)$	$O((m+n)/B)$	$O((m+n)/B)$	$O((m+n)/B)$

TABLE I: Cache behavior of various sparse graph and matrix operations. $n = |V|$, $m = |E|$. The table lists various graph representations and the algorithmic runtime of common graph operations in the external memory model due to Aggarwal and Vitter [AV88] where B is the cache line (or disk block) size. The RAM model (without cache analysis) is the special case where B or $\lg(B)$ is 1. We analyze Packed CSR in the right-most column. * We use a c++ vector for our implementation of CSR, so we do not need to rebuild the node list every time we add a node.

based on the packed memory array (PMA) [BH07]. PCSR retains the locality and ordering of CSR, admitting fast searches and traversals with efficient cache usage, while supporting fast inserts. Table I lists the cache behavior and space/time guarantees of basic operations for popular graph storage formats and PCSR.

- We implemented PCSR, adjacency list, blocked adjacency list, and CSR. We find that PCSR supports inserts orders of magnitude faster than CSR and is about a factor of two slower for traversal benchmarks such as sparse matrix-vector multiplication.

The rest of this paper is organized as follows: Section II reviews graph storage formats, Section III reviews the theoretical guarantees of the PMA data structure, and Section IV reports the results of a variety of benchmarks using the different data structures. We summarize our results in Section V and suggest future work.

II. Graph Storage Formats

In this section, we describe the following graph storage formats: adjacency matrix, adjacency list, blocked adjacency list, and CSR. We detail their respective space/time tradeoffs in Table I. For a graph $G = (V, E)$, we denote the number of nodes by $n = |V|$ and number of edges by $m = |E|$.

Adjacency Matrix

An ***adjacency matrix*** is the most basic graph storage format. It stores an $n \times n$ matrix for a graph of n nodes. The entry at $[u, v]$ corresponds to the value of the edge (u, v) (or has 0 if the edge does not exist). It excels in storing dense graphs because it does not store any pointers and therefore minimizes overhead if the graph is almost fully connected.

The adjacency matrix wastes space when the graph is sparse because it requires n^2 space. Furthermore, adding nodes requires rebuilding the entire data structure. Finally, sparse graph traversals on adjacency matrices require iterating over the entire matrix of size n^2. Since the number of edges is $m \ll n^2$ for many sparse graphs, a graph traversal using an adjacency matrix is not work efficient.

Adjacency List

Another common sparse graph storage format is the ***adjacency list*** (AL). Adjacency lists keep an array of nodes where each entry stores a pointer to a linked list of edges. The pointer at index u in the node list points to a linked list where each element v in the linked list is an outgoing edge (u, v).

Adjacency lists support fast inserts but have high space overhead and slow searches because the edges are unsorted. Adjacency lists also exhibit poor cache behavior because they lack locality. A variant of adjacency lists called ***blocked adjacency lists*** (BAL) uses blocks to store edges. Blocked adjacency lists exhibit faster traversals because of improved locality but require extra space for extremely sparse graphs. Blandford, Blelloch, and Kash [BBK04] introduced a dynamic graph data structure based on BAL with many constant-factor improvements but stop short of giving theoretical guarantees. For simplicity, we compare PCSR with standard adjacency lists of various block sizes.

Figure 1 shows an example of a graph stored in adjacency list format.

Compressed Sparse Row

Compressed sparse row (CSR) is a popular format for storing sparse graphs and matrices. It efficiently packs all the entries together in arrays, allowing for quick traversals of the data structure.

978-1-5386-5989-2/18/$31.00 ©2018 IEEE

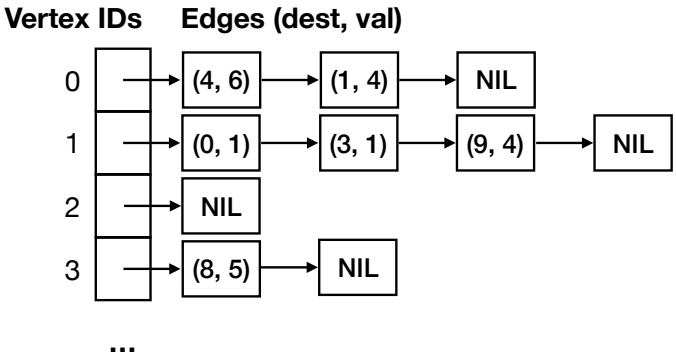


Fig. 1: An example of a graph stored in an adjacency list. Each entry in the nodes array points to a linked list of edges. The vertex ID in the nodes array implicitly stores the source. For weighted graphs, we store a tuple of destination vertex and edge value for each edge.

CSR uses three arrays to store a sparse graph: a node array, an edge array, and a values array[1]. Each entry in the node array contains the starting index in the edge array where the edges from that node are stored in sorted order by destination. The edge array stores the destination vertices of each edge. CSR stores a graph $G = (V, E)$ in size $O(|V| + |E|)$, but needs to be rebuilt upon any changes. Figure 2 contains an example of a graph stored in CSR format.

Inserting an edge into a graph in CSR format takes time linear in the size of the graph in the worst case. To insert an edge (u, v) into a graph in CSR format, we first must search all edges with source vertex u to find the edge with the smallest destination larger than v. Then we insert (u, v) into the edge list and slide all elements after it over by one. We then increment the elements in the node array for all vertices greater than u by one. The entire edge array may need to be resized and copied into a larger block of memory if there are too many elements in the structure.

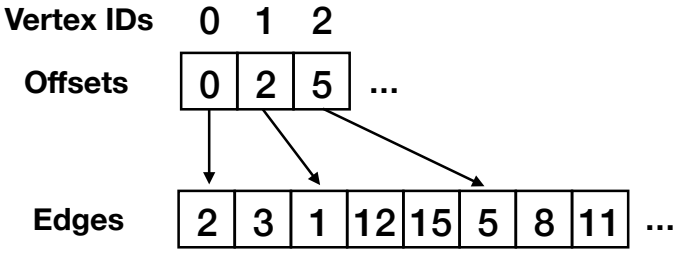


Fig. 2: An example of an unweighted graph stored in compressed sparse row. The values stored in the edges array represent the destination. The vertex ID in the offset array implicitly stores the source. For weighted graphs, there is an additional values array.

Pinar and Heath [PH99] introduced a variant of CSR called Blocked Compressed Sparse Row (BCSR), where the locations of nonzero blocks are stored in CSR format. For our experiments, we focus on unblocked CSR for simplicity.

[1]not needed in the unweighted case

Practitioners often use CSR for storing social networks and random graphs, which we focus on in this paper.

III. Packed Compressed Sparse Row

In this section, we review the structure and theoretical properties of the packed memory array (PMA) [BH07]. The PMA maintains edges in sorted order and leaves space between elements to support fast inserts and deletes.

Packed Memory Array

The PMA stores N items in an ordered list of size $O(N)$ and supports inserts and deletes in $O(\lg^2(N))$.

At a high level, the PMA avoids changing the entire data structure after each insert or delete by maintaining spaces between elements and rebalancing when there are too many or too few elements in a range. It maintains spaces of size $O(1)$ between groups of elements of size $O(1)$ to enable easy insertions and deletions. The PMA maintains these spaces by rebalancing the structure and redistributing the elements whenever a section of the data structure becomes too sparse or too dense. Specifically, if the size of the data structure at some time t is n_t, the PMA keeps an implicit tree with $n_t/\lg(n_t)$ leaves of the data structure where each leaf has $\lg(n_t)$ slots. The density of a node is determined by the number of elements in it divided by the total number of slots in the node. If the density is too low or too high, the PMA rebalances the elements across a child or parent, respectively. If the entire array becomes too dense or too sparse, we resize the entire data structure.

Figure 3 shows an example of the implicit binary tree on the PMA intervals. If an interval becomes too dense, we walk up the tree and redistribute when we find an interval that is appropriately dense.

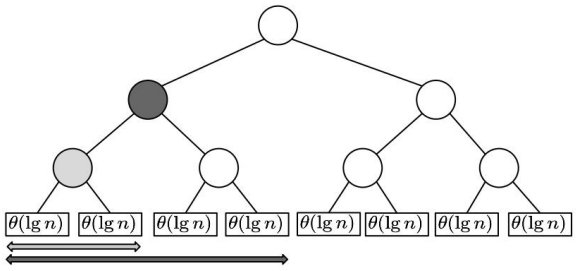


Fig. 3: An example of the implicit binary tree on the PMA intervals. If we insert a new element in a leaf and the corresponding interval becomes too dense (shown in light grey), we walk up the tree until we find an interval with a density in the allowed range (shown in dark grey). In the worst case, we walk up to the root and do a rebalance of the entire PMA. This figure is from [Dem12].

Description of Structure

PCSR uses the same vertex and edge lists as CSR but uses a PMA instead of an array for the edge list. Adding both edges and nodes to the graph requires updates to both the vertex and edge lists. We use a c++ vector with doubling and halving for

978-1-5386-5989-2/18/$31.00 ©2018 IEEE

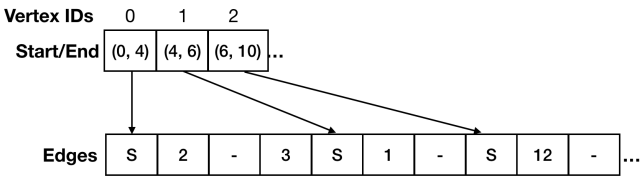


Fig. 4: An example of a graph stored in PCSR. S denotes the sentinels. The ranges (start, end) in the vertex array denote the start and end of the corresponding edges in the edge array.

the node list. Each element in the node list stores start and end pointers into the edge list for its range. Each nonempty entry in the edge list contains the destination vertex and the edge value. Each node's range in the edge list has a corresponding sentinel entry in the edge list which points back to the source in the node list for updating the node pointers.

We present an example of a graph stored in PCSR format in Figure 4.

The size of the node list is $O(n)$ since it stores two pointers for each node. The size of the edge PMA is $O(n+m)$ since it stores an entry for each edge and node. The size of an PMA is $O(N)$ where N is the number of elements in the PMA. Therefore, the total space usage of PCSR is $O(n+m)$, the same as standard CSR.

Operations

Adding a Node. We add nodes by extending the length of the node array by one with a pointer to the end of the edge structure. We then add the sentinel edge into the edge structure.

Adding an element to the end of the node structure is $O(1)$ and adding an element to the edge structure is $O(\lg^2(n+m))$, so the overall time is $O(\lg^2(n+m))$.

Adding an Edge. Adding an edge first requires finding the node in the node array, then requires a binary search on the relevant section of the edge array to insert the edge in sorted order indexed by its destination. If a rebalance is triggered, we check every moved edge to see if it is a sentinel. If so, we update the node array with its new location.

Finding the location in the node structure is $O(1)$, binary searching the relevant section of the edge array is $O(\lg(deg(v)))$, and inserting is $O(\lg^2(n+m))$, giving us $O(\lg^2(n+m))$ for the overall time.

Removing an Edge. Removing an edge is symmetric to adding an edge. We find the edge with binary search and then remove it from the PMA and rebalance if necessary. Therefore, the runtime is the same as adding an edge: $(O(\lg^2(n+m)))$.

Removing a Node. First, we set the start and end pointers into the edge array to null. We can also keep track of the number of removed nodes and rebuild the entire structure when the number of non-removed nodes equals the number of removed nodes. Nodes can only be removed after all of their edges have been removed[2]. We need to both mark the node in the node structure and remove the sentinels from the edge structure. This takes time $O(\lg^2(n+m))$.

[2]It would be possible to implement a faster bulk edge removal by deleting all the edges at once and not doing rebalances until the end.

To maintain the node list with $O(n)$ entries we can simply rebuild the structure every time the number of removed nodes exceeds half the number of nodes before node deletions.

We have not implemented removing edges and nodes, but their asymptotic performance is symmetric to adding edges and nodes.

IV. Results

We evaluated PCSR against CSR, adjacency list (AL), and blocked adjacency lists (BAL). We do not compare to the adjacency matrix due to its inability to scale to large graphs. We will evaluate the structures on their performance and their space usage. We focus on the sparse case since the adjacency matrix outperforms all other graph representations if the graph is dense. We randomly generated variable numbers of edges in a graph with a constant number of nodes for our tests.

System

We ran our experiments on an AWS instance with 18 cores, with hyperthreading, and 2.9GHz clock speed. The machine had 64GB of RAM, 32K of L1 cache, 256K of L2 cache, and 25600K of L3 cache. Programs were written in c++ and compiled with GCC 4.8.5 with -O3. All programs were run sequentially.

Memory Footprint

We measured the memory footprint of each data structure for a fixed number of nodes and variable number of edges. Figure 5 shows the relative growth of the memory footprint of each graph representation.

The BALs use much more size than necessary when the average degree is small because most of the space in the blocks is empty.

The `c++` vector for the edge list in CSR doubled the speed of inserts since our implementation of CSR (on average) only needs to copy half of the elements and not all of them on each insert. Therefore, we also compare to the ideal CSR size without extra space.

We found that there is about a factor of 2 between the size of an ideal CSR (without extra padding) and the worst AL and that PCSR only has a space overhead of between 20% and 30%.

Inserts

We benchmarked the time to insert unique edges on all of the data structures. We generated edges uniformly at random without replacement. Figure 6 shows the time to insert $100,000$ edges with a fixed number of nodes and a variable number of edges. AL-based representations supported fast inserts, while CSR was the slowest. CSR starts about 3 orders of magnitude slower than all other representations and also scales much worse. Therefore, we are unable to run it for large numbers of edges. We find that in practice that PCSR is about 3-4 times slower than AL based representations.

978-1-5386-5989-2/18/$31.00 ©2018 IEEE

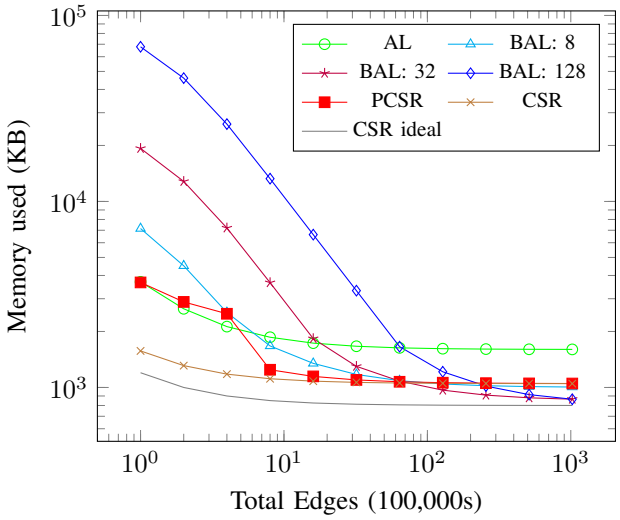


Fig. 5: Size per 100,000 edges of each data structure with 100,000 nodes and a variable number of edges. The x-axis represents the number of edges, while the y-axis represents the size per 100,000 elements.

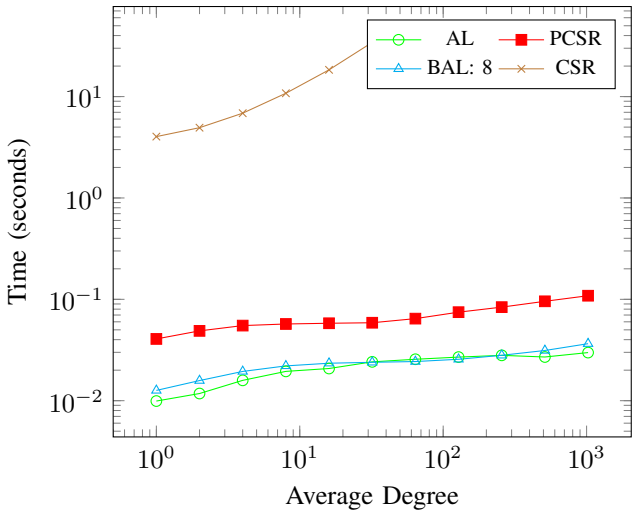


Fig. 6: Time to insert 100,000 edges with a fixed number of nodes. We used 100,000 nodes and a variable number of total edges added.

Updates

We benchmarked update operations on all of the data structures. We generated edges uniformly at random with replacement. Figure 6 shows the time to insert edges that potentially exist in the edge list with a fixed number of nodes. The difference between update and insert is that update requires a search beforehand to check if the edge is already in the structure. We again show the time for updating (or inserting) $100,000$ edges.

PCSR outperformed all other structures when the average degree grew to reasonable sizes, as expected from Table I. Once again, CSR is several orders of magnitude worse and is too slow to complete on reasonable input sizes. Additionally,

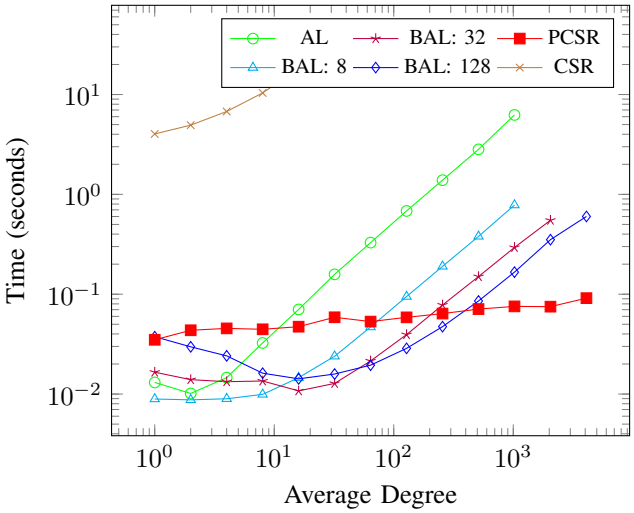


Fig. 7: Time to insert or update 100,000 edges with a fixed number of nodes. We used 100,000 nodes and added a variable number of edges.

the AL-based representations take linear time to search and take much longer than the $O(\lg^2(n))$ search time of PCSR. While the high search cost in AL-based representations can be somewhat offset by increasing the size of the block, larger block sizes increase the size of the AL and slow insertions.

Sparse Matrix-vector Multiplication

Figure 8a shows the time to perform a sparse matrix-vector multiplication using the different structures with 100,000 nodes and a variable number of edges.

Although the asymptotic complexity for SpMV is the same for all of the structures, the AL-based structures can suffer from poor cache behavior. Increasing the block size in BALs can improve cache performance. PCSR avoids the problem of cache locality because it stores all of its edges in a single array. SpMV takes longer in AL than PCSR because the PCSR has better cache behavior. SpMV in PCSR is within a factor of 2 and often within 20% of SpMV in CSR.

PageRank and BFS

Figure 8b shows the time to perform an iteration of PageRank using the different structures with 100,000 nodes and a variable number of edges.

Figure 8c shows the time to compute the distance to each node from a randomly chosen source node using each of the different structures with 100,000 nodes and a variable number of edges.

The time to perform a BFS and an iteration of PageRank scales with the number of edges in the graph in all representations.

PCSR achieved within 25% of CSR's runtime on most input sizes. CSR was the fastest, followed by PCSR and BAL-128. BAL with bigger blocks would perform even better (closer to CSR).

978-1-5386-5989-2/18/$31.00 ©2018 IEEE

Graph Format	AL	BAL 8	BAL 32	BAL 128	CSR
Slashdot					
Size	0.88	0.82	1.47	4.71	0.41
SpMV	10.87	1.29	1.45	1.32	0.39
BFS	8.86	1.20	1.42	1.17	0.47
PageRank	13.38	1.64	1.85	1.72	0.36
Adding edges	0.25	0.25	0.25	0.25	525.00
Updating edges	10.50	1.25	1.00	0.75	508.75
Pokec					
Size	0.93	0.71	0.98	2.75	0.45
SpMV	15.95	2.43	1.21	1.17	0.51
BFS	7.25	1.64	1.02	1.00	0.48
PageRank	11.77	3.04	1.78	1.72	0.54
Adding edges	0.25	0.50	0.25	0.25	31628.50
Updating edges	9.17	2.50	0.83	0.67	21005.83
Livejournal					
Size	1.05	0.87	1.36	4.00	0.49
SpMV	20.40	2.77	2.20	2.10	0.59
BFS	9.55	2.30	1.34	1.40	0.53
PageRank	16.20	5.36	2.40	2.73	0.54
Adding edges	0.25	0.25	0.25	0.50	70787.00
Updating edges	13.17	4.00	1.50	1.17	46835.00

TABLE II: Real-world graphs. We tested on Slashdot, pokec, and Livejournal. All times are normalized against PCSR.

Real World Graphs

We also tested on three social network graphs of varying sizes from the Stanford Large Network Dataset Collection and report our results in Table II. They were Slashdot, with 77,360 nodes and 905,468 edges, Pokec with 1,632,803 nodes and 30,622,564 edges, and LiveJournal with 4,847,571 nodes and 68,993,773 edges.

For adding and updating edges, we added $1,000$ random edges chosen without replacement with the same distribution as the edges in the original graph.

We found that PCSR was about a factor of 2 slower than CSR on graph computations but had much faster updates. The AL-based representations had similar size to PCSR and were between 2 to 10 times slower on graph computations but about 4 times faster in adding edges.

V. Conclusion

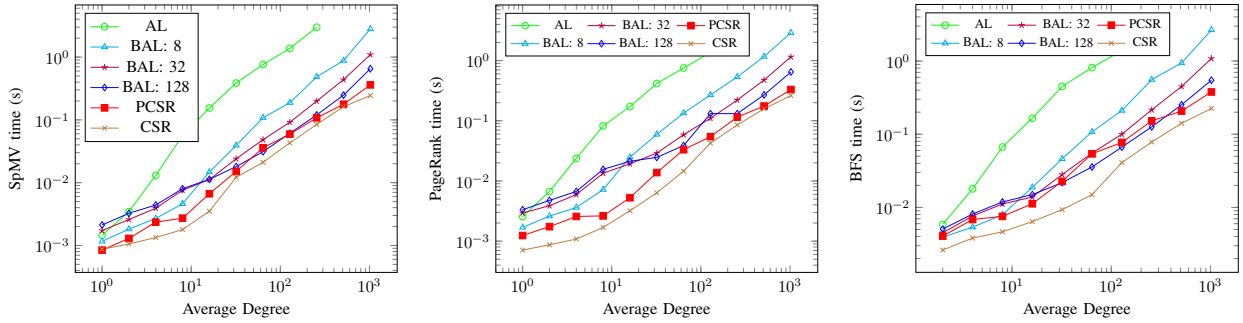
We have implemented PCSR, a dynamic graph storage format based on the packed memory array. We find that for slightly more storage and query time, we are able to achieve similar mutability speeds to the adjacency list. CSR was unable to handle many inserts in a reasonable amount of time. PCSR was orders of magnitude faster for inserts and updates than CSR and adjacency list while maintaining similar graph traversal times.

The growth of social networks and other changing graphs necessitates the need for efficient dynamic graph structures. PCSR is a basic dynamic graph storage format that can fit into existing graph processing frameworks and support fast insertions with comparable traversal times.

Future work includes implementing deletes, other graph algorithms such as shortest-paths, and PCSR in external-memory. Additionally, we may be able to speed up PCSR with a parallel implementation.

Acknowledgments

We would like to thank Julian Shun for his helpful comments and suggestions.



(a) Sparse Matrix- Vector Multiplication (b) PageRank (c) BFS

Fig. 8: Time with 100,000 nodes and a variable number of edges. The x-axis represents the number of edges, while the y-axis represents the time

978-1-5386-5989-2/18/$31.00 ©2018 IEEE

REFERENCES

[AV88] Alok Aggarwal and Jeffrey S. Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, September 1988.

[BBAB+09] David A Bader, Jonathan Berry, Adam Amos-Binks, Charles Hastings, Kamesh Madduri, and Steven C Poulos. Stinger: Spatio-temporal interaction networks and graphs (sting) extensible representation. 2009.

[BBK04] Daniel K Blandford, Guy E Blelloch, and Ian A Kash. An experimental analysis of a compact graph representation. 2004.

[BH07] Michael A Bender and Haodong Hu. An adaptive packed-memory array. *ACM Transactions on Database Systems (TODS)*, 32(4):26, 2007.

[Dem12] Erik Demaine. Lecture notes in advanced data structures, March 2012.

[DH11] Timothy A. Davis and Yifan Hu. The university of Florida sparse matrix collection. *ACM Transactions on Mathematical Software*, 38(1):1–25, November 2011.

[EMRB12] David Ediger, Robert McColl, Jason Riedy, and David A Bader. Stinger: High performance data structure for streaming graphs. In *High Performance Extreme Computing (HPEC), 2012 IEEE Conference on*, pages 1–5. IEEE, 2012.

[GM12] Sudipto Guha and Andrew McGregor. Graph synopses, sketches, and streams: A survey. *Proceedings of the VLDB Endowment*, 5(12):2030–2031, 2012.

[KBG12] Aapo Kyrola, Guy E Blelloch, and Carlos Guestrin. Graphchi: Large-scale graph computation on just a pc. USENIX, 2012.

[KGKM16] James King, Thomas Gilray, Robert M Kirby, and Matthew Might. Dynamic sparse-matrix allocation on gpus. In *International Conference on High Performance Computing*, pages 61–80. Springer, 2016.

[MMMS15] Peter Macko, Virendra J Marathe, Daniel W Margo, and Margo I Seltzer. Llama: Efficient graph analytics using large multiversioned arrays. In *Data Engineering (ICDE), 2015 IEEE 31st International Conference on*, pages 363–374. IEEE, 2015.

[PH99] A. Pinar and M. T. Heath. Improving performance of sparse matrix-vector multiplication. In *Supercomputing, ACM/IEEE 1999 Conference*, page 30, November 1999.

[Saa03] Y. Saad. *Iterative methods for sparse linear systems*. SIAM, Philadelphia, 2nd ed edition, 2003.

[Say] David Sayce. 10 billions tweets, number of tweets per day. http://www.dsayce.com/social-media/10-billions-tweets/. Accessed: 2018-05-09.

[Shi91] Olin Shivers. *Control-flow analysis of higher-order languages*. PhD thesis, 1991.

[SLHT17] Mo Sha, Yuchen Li, Bingsheng He, and Kian-Lee Tan. Accelerating dynamic graph analytics on gpus. *Proceedings of the VLDB Endowment*, 11(1):107–120, 2017.

[UKBM11] Johan Ugander, Brian Karrer, Lars Backstrom, and Cameron Marlow. The anatomy of the facebook social graph. *arXiv preprint arXiv:1111.4503*, 2011.

[WKF+15] Charith Wickramaarachchi, Alok Kumbhare, Marc Frincu, Charalampos Chelmis, and Viktor K Prasanna. Real-time analytics for fast evolving social graphs. In *Cluster, Cloud and Grid Computing (CCGrid), 2015 15th IEEE/ACM International Symposium on*, pages 829–834. IEEE, 2015.

[XG04] Wenpu Xing and Ali Ghorbani. Weighted pagerank algorithm. In *Communication Networks and Services Research, 2004. Proceedings. Second Annual Conference on*, pages 305–314. IEEE, 2004.

978-1-5386-5989-2/18/$31.00 ©2018 IEEE