A Fast Data Structure for Dynamic Graphs Based on Hash-Indexed Adjacency Blocks

Alexander van der Grinten
Humboldt-Universität zu Berlin, Germany

Maria Predari
Humboldt-Universität zu Berlin, Germany

Florian Willich
Humboldt-Universität zu Berlin, Germany

Abstract

Several dynamic graph data structures have been proposed in literature. Yet, these data structures either offer limited support for arbitrary graph algorithms or they are designed as part of specific frameworks (e.g., for GPUs or specialized hardware). Such frameworks are difficult to adopt to arbitrary graph computations and lead practitioners to fall back to less sophisticated solutions when dealing with dynamic graphs. In this work, we propose a new "dynamic hashed blocks" (DHB) data structure for sparse dynamic graphs and matrices on general-purpose CPU architectures. DHB combines an efficient block-based memory layout to store incident edges with an additional per-vertex hash index for high degree vertices. This hash index allows us to quickly insert edges without introducing duplicates, while the block-based memory layout retains advantageous cache locality properties of traditional adjacency arrays.

Experiments show that DHB outperforms competing dynamic graph structures for edge insertions, updates, deletions, and traversal operations. Compared to static CSR layouts, DHB exhibits only a small overhead in traversal performance. DHB's interface is similar to general-purpose abstract graph data types and can be easily used as a drop-in replacement for traditional adjacency arrays. To demonstrate that, we modify the well-known NetworKit framework to use DHB instead of its own dynamic graph representation. Experiments show that this modification only slightly penalizes the performance of graph algorithms while considerably boosting update rates.

2012 ACM Subject Classification Theory of computation → Dynamic graph algorithms

Keywords and phrases dynamic graph data structures, sparse matrix layout, dynamic algorithms, parallel algorithms, graph analysis

Digital Object Identifier 10.4230/LIPIcs.SEA.2022.11

Supplementary Material *Software (Source Code)*: `https://github.com/hu-macsy/dhb`
archived at `swh:1:dir:9576f651115c810985803de5214f519bfc9600ef`

Funding *Alexander van der Grinten*: The author was supported by German Research Foundation (DFG) grant GR 5745/1-1 (DyANE).
Maria Predari: The author was supported by German Research Foundation (DFG) DFG grant ME 3619/4-1 (ALMACOM).

Acknowledgements The authors would like to thank Duy Le Thanh for his help in setting up some competitors.

1 Introduction

Large-scale graph data are ubiquitous in various areas of science and engineering [1, 16]. Yet, their efficient processing is still challenging. For one, the graphs in question are large and sparse. Typically, the number of neighbors [non-zero values] of a vertex [row/column] in a sparse graph [matrix] is bounded by a small constant and most possible edges [matrix entries]

do not exist [are zero].[1] Thus, sparse-friendly data structures are employed to avoid wasting processing and memory on empty entries. Dynamic graph applications introduce a second challenge as, in practice, most well-known graph processing frameworks still use static data structures [17, 26, 27]. Static graph representations are memory-efficient and support fast operations but lack flexibility in terms of dynamic updates. Examples of highly dynamic data are the Facebook and Twitter graphs, where users and connections are added/removed continuously [1]. Those mutations on the data structure (updates) are typically followed by graph queries to ensure consistency of analytics.

Recently, a number of dynamic graph frameworks were proposed, enabling graph processing and analysis for dynamically changing data [5, 8, 30, 31]. These frameworks often perform updates and queries concurrently, either via snapshots (simultaneous graph copies) or via batched updates. Existing dynamic graph frameworks differ in multiple aspects: their design, concurrency strategy, API support for graph analytics and applicability to generic architectures. In general, there are two performance goals for dynamic graph algorithms. The first goal aims at maximizing update rates, while maintaining low memory utilization. The second one aims at accelerating graph analytics running on top of their graph structure. Most dynamic graph frameworks consider the first goal but ignore the second. These frameworks focus mainly on graph updates and offer limited support for developing higher-level graph analytics [30]. Moreover, the ones that also consider the second goal are systematically slower than the ones that only focus on the first [19].

We propose a new data structure (DHB) for efficient processing of dynamically mutating large-scale sparse graphs. DHB is designed for general-purpose CPU architectures and combines an efficient block-based memory layout to store incident edges with an additional hash index for high degree vertices. The data structure is conceptually simple (and straightforward to implement); yet, we are not aware of any systematic experimental evaluation of this (or an equivalent) data structure. DHB utilizes on average the same memory as NetworKit, a static graph data framework using adjacency arrays. In a single-threaded environment, DHB outperforms all competitors regarding insertions, deletions and weight updates for different graph types and sizes, being on average from 1.9 to 93.8 × faster. Our data sets include static and temporal graphs with up to 1.8B edges. In a parallel environment, DHB's performance is similar to aspen which is reported to be one of the fastest dynamic graph frameworks [4, 7]. Moreover, DHB implements efficient lookups and other graph operations to accelerate common graph algorithms. Our experiments demonstrate that running BFS in a dynamic setting on top of DHB outperforms the corresponding BFS execution on top of aspen by a factor of 2.5. Finally, we demonstrate that the overhead of using DHB instead of a static graph structure is low. More precisely, we integrate DHB as a drop-in replacement into NetworKit and run BFS in a static setting (no updates). We observe that BFS on top of DHB is only 15% slower than BFS on top of NetworKit's adjacency arrays. This is a low overhead compared to the significant performance improvement under edge updates.

The paper is organized as follows: in Section 2, we present relevant background on traditional graph structures and in Section 3 we briefly review existing solutions for dynamic graphs. In Section 4, we present our newly proposed graph structure (DHB) while in Section 5 we evaluate it against both static and dynamic competitors. Finally, in Section 6 we give our concluding remarks.

[1] Due to the correspondence of matrices and graphs [11], in the remainder, we use these terms interchangeably.

2 Preliminaries

Static graph representation

We consider directed sparse graphs $G = (V, E)$. Undirected graphs are modelled by splitting each undirected edge $\{u, v\}$ into two directed ones. General graphs are commonly represented by their $n \times n$ adjacency matrix A, where $n = |V|$ (the number of non-zero entries of A correspond to the number of edges $m = |E|$). The data structures that we consider, store A into a sparse layout. Common sparse data structures are the adjacency array (or adjacency list), the coordinate list (COO) and the compressed sparse row (CSR) (or compressed sparse column). In the adjacency array representation, each vertex u has an associated array that maintains the IDs of all vertices in its neighborhood $N(u)$. Adjacency arrays use $\mathcal{O}(n + m)$ space and check connectivity of two vertices in $\mathcal{O}(\deg(u)) \subseteq \mathcal{O}(\deg_{\max})$ time, where $\deg(u)$ is the degree of u and $\deg_{\max}$ is the maximum degree in G. COO holds an array of (row, column, value) tuples and is similar to the adjacency array in the asymptotic time, space complexity, and general design. The main difference is that each edge is stored explicitly, with both its source and destination vertex. The above storage formats allow for a limited number of updates but have big overheads due to re-allocation of data and slow search operations. Finally, CSR uses three arrays to store a sparse graph: a node array, an edge array, and a values array. 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 in $\mathcal{O}(n + m)$ space. Inserting an edge into the CSR format takes linear time in the worst case. The entire edge array may need to be copied into a larger block of memory if there are too many elements in the structure. As a result, CSR is also not a suitable format for dynamic updates.

Updates

When talking about dynamic graphs, we consider edge updates, i.e., insertion and deletions of edges, as well as edge weight changes. Insertions and/or deletions of vertices are handled by standard techniques, i. e. by resizing the data structure to be able to hold enough vertex IDs and by storing an additional bit per vertex to determine if the vertex is deleted or not.

Dynamic challenges

For dynamic graph structures there is a trade off between efficient updates, optimal memory layout, and fast lookups. Traditional adjacency arrays allocate one array per vertex. *Block-based* data structures refine this strategy by storing incident edges in blocks. A block is an array whose size comes from a set of size classes (often powers of two). Blocks of the same size are stored in a superblock of fixed size (e.g., 2 MiB). On the other hand, *compressed sparse* layouts simply concatenate all adjacency arrays to a single memory allocation. This makes it easy to iterate over edges, but difficult to resize the data structure. Finally, other data structures do not store edges in arrays at all but opt for a different primitive, such as hash tables. These data structures usually exhibit considerable overhead when iterating over incident edges. Regarding edge lookups (e. g. to detect duplicates or to update edge weights), many implementations simply loop over the entire adjacency list. Other data structures *sort* the edges to support $\mathcal{O}(\log n)$ lookups; however, this comes with the cost of more expensive dynamic updates (i. e. $\mathcal{O}(\deg_{\max})$ to delete an arbitrary edge). Implementations based on hash tables can usually look up edges in expected $\mathcal{O}(1)$ time.

3 Related Work

Recently, a number of dynamic graph frameworks have been proposed in the literature. We focus on frameworks that are designed for CPU architectures. STINGER [8] is a dynamic graph structure for multi-core architectures that stores the adjacency information of each vertex using blocked linked lists of pre-selected, fixed size. GraphIn [24] allows for incremental graph processing by combining two static graph data structures: CSR for the original input and a COO to store new edge updates. The framework has similar limitations as COO; it is constrained to a limited number of updates (pre-defined by the users). PSCR [28] (later extended to PPCSR [29]) is a dynamic data structure based on the packed memory array (PMA) [3]. Sha et al. [25] also uses a variant of PMA. PMA is an array with all neighborhoods (i.e., essentially a CSR) augmented with an implicit binary tree structure that enables edge insertions and deletions in $\mathcal{O}(\log^2 n)$ time. Unfortunately, the above solutions can not be easily integrated into existing graph frameworks, due to their limited support for arbitrary graph operations. Moreover, ASPEN [7] uses a novel probabilistic tree called a C-tree to store the graph structure and is reported to be one of the faster frameworks targeting CPUs for insertions and deletions [4]. Recently, TERRACE [20] introduced a hierarchical graph structure for dynamic graphs that uses both arrays and trees to store adjacencies, depending on their size.

Although our focus is on solutions for general-purpose CPU architectures, we briefly describe HORNET [5], a data structure designed for GPU architectures. HORNET is relevant to our work as it uses a similar block-based mechanism. More precisely, HORNET groups adjacency information of several vertices together in blocks, whose sizes are a power of two. Additionally, it uses a vectorized bit tree, and B+trees for managing memory blocks. HORNET is shown to outperform competing dynamic graph structures implemented for GPU architectures [4, 5]. An excellent survey on dynamic graph data structures, frameworks and databases can be found in [4].

4 Dynamic Hashed Blocks (DHB)

Our new DHB data structure uses a block-based memory layout with an additional hash index for high degree vertices to accelarate lookups of neighbors. While DHB builds on simple algorithmic primitives, the resulting data structure is highly competitive with state-of-the-art graph data structures, as demonstrated by our experiments in Section 5. Our data structure is designed around the following properties:

Basic operations. DHB supports the usual operations expected from an abstract data type for (dynamic) graphs: changing the number of vertices, insertion, update, and deletion of edges, edge existence queries, as well as neighborhood traversals. In contrast to frameworks such as Ligra or ASPEN [7], our data structure allows direct access to the neighbors of a vertex (while these frameworks only allow access via an EDGEMAP function). Unlike many other dynamic graph frameworks, we do *not only* focus on batch updates.

Random access. Likewise, many algorithms expect the ability to access the i-th neighbor of vertex u, where $i \in [0, \deg(u))$. For example, this is frequently used to sample a random neighbor of a vertex. To support this operation in $\mathcal{O}(1)$, it is convenient to store the neighbors of a vertex in a contiguous array. This requirement motivates the use of a block-based storage format in DHB.

Arbitrary neighbor ordering. Some algorithms require neighborhoods to be ordered in specific ways to work correctly. For example, the SUITOR algorithm to approximate the weighted matching problem requires edges to be sorted according to non-increasing edge

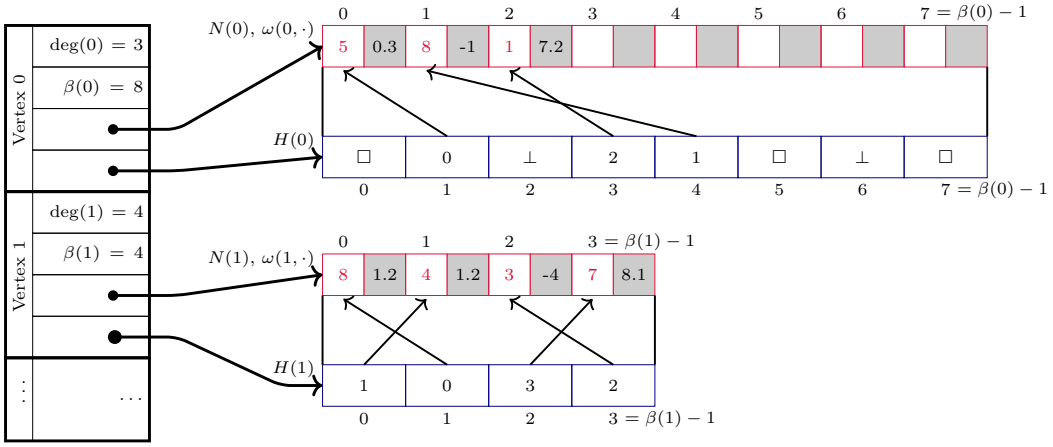


Figure 1 Layout of DHB. Each vertex has an associated adjacency block consisting of $N(\cdot)$ (red) and $H(\cdot)$ (blue). The adjacency block of vertex 0 has five empty slots left, while the adjacency block of vertex 1 is fully occupied. Gray boxes indicate edge weights that are stored interleaved with $N(\cdot)$.

weights [18]. To support these algorithms, the graph data structure should not impose a fixed order on the neighbors of each vertex, but preserve a user-specified order. This makes it possible to support an operation to re-order edges according to an arbitrary order. These requirements essentially rule out data structures that store neighbors directly in hash tables and data structures that rely on sorting to efficiently look up neighbors.

Support for concurrency. Many parallel algorithms (e. g. parallel graph generation algorithms) expect that neighbors of different vertices can be mutated in parallel. For this reason, we choose to use per-vertex hash indices (and not a global edge index) to accelerate lookups of neighbors.

4.1 Neighbors and Hash Index

The layout of DHB's main data structure is illustrated in Figure 1. DHB associates four data fields with each vertex u: (i) the current degree $\deg(u)$, (ii) a non-negative integer $\beta(u)$ that will store the number of neighbors currently reserved for vertex u, (iii) a pointer to an array $N(u)$ that can hold up to $\beta(u)$ neighbors of u, and (iv) a pointer to the hash index $H(u)$ of u. These fields are stored in an array indexed by the vertex ID u.[2] We define an *adjacency block* as the combination of the array that holds $N(u)$ and the array that holds $H(u)$. We call $\beta(u)$ the block size of the adjacency block of u. $\beta(u)$ will always be a power of two to accelerate the maintenance of the hash index. By extending the data structure appropriately, we always guarantee that $\deg(u) \leq \beta(u)$ (see below for details). In particular, the first $\deg(u)$ entries of $N(u)$ always hold the current neighbors of u, while the remaining $(\beta(u) - \deg(u))$ entries remain empty until new neighbors are inserted.

When edge weights and/or other per-edge data needs to be stored, we store this data interleaved with $N(u)$ (i. e. using an "array of structures", the gray boxes in Figure 1). This minimizes the number of pointers that our data structure has to store per vertex. For algorithms that only rarely access associated data, the data structure can be modified to

[2] Without loss of generality, we assume that vertices are identified by non-negative integer IDs in the range $[0, n)$. If this assumption is not satisfied (e. g. due to large gaps between IDs), an additional hash map can be used to map input vertex IDs to internal vertex IDs.

store pointers to additional per-vertex arrays that hold data associated with edges (yielding a "structure of arrays"). Another implementation strategy is assigning an edge ID to each edge, and storing associated data in a separate array that is indexed by edge ID (e. g. this is what NetworKit does).

The hash index $H(u)$

The hash index of a vertex u is used to quickly look up the position of neighbors of u in the adjacency array $N(u)$. We maintain $H(u)$ only for high degree vertices; for low-degree vertices, it is more efficient in practice to simply scan the entire adjacency list to find the index of a neighbor. We define high degree vertices as those whose neighborhood spans several cache lines (in our experiments, this threshold is set to 16 cache lines). If $H(u)$ is present, it consists of an array of $\beta(u)$ non-negative integers. $H(u)$ is used to implement a hash table based on open addressing. We use standard linear probing to resolve collisions. However, our hash table does not directly store any graph data; instead, it stores indices into $N(u)$. Initially, all slots of $H(u)$ are set to a special value $\square$ to indicate that the slots are empty. Another special value $\perp$ is used to indicate slots that became empty after deletions (i. e. $\perp$ represents a tombstone). If $H(u)[j] \notin \{\square, \perp\}$, then $H(u)[j]$ is always a valid index into $N(u)$, i. e. $H(u)[j] \in [0, \deg(u))$. We say that a slot j of the hash index corresponds to neighbor v of u if $N(u)[i] = v$, where $i = H(u)[j]$. The hash index will be constructed such that each non-empty slots of $H(u)$ correspond exactly to the neighbors of u. In Figure 1, this is represented by the arrows from $H(u)$ to $N(u)$.

Since the operations of DHB depend on the correct maintenance of the hash index, we briefly discuss how operations on the hash index itself behave. Looking up the possible neighbor v in the hash index of vertex u proceeds as follows: we start by evaluating a hash function $h : V \to \mathbb{N}$ to probe $H(u)$ at $j := (h(v) \mod \beta(u))$. Since $\beta(u)$ is a power of two, the modulo operation can be computed by a simple bitwise AND. If $H(u)[j] = \square$, then v is not a neighbor of u. In case we want to insert v, we can now set $N(u)[i] \leftarrow v$ and $H(u)[j] \leftarrow i$, where i is the smallest previously unused index of $N(u)$. On the other hand, if $H(u)[j] = \perp$, we increment $(j \mod \beta(u))$ (i. e. we probe linearly). Otherwise, $H(u)[j] \notin \{\square, \perp\}$. Let $i = H(u)[j]$. We can assume that i is a valid index into $N(u)$. If $N(u)[i] = v$, then v is a neighbor of u and we found its index i into $N(u)$. Otherwise, we continue probing linearly by incrementing $(j \mod \beta(u))$ and repeating the procedure.

We note that when updating $H(u)$ during the insertion of a new neighbor v, we can overwrite the first slot j with $H(u)[j] = \perp$, if we encounter such a slot; however, to correctly detect duplicates, we first have to finish through the entire probe sequence (i. e. until we either find v or $H(u)[j] = \square$). Furthermore, we remark that we can periodically purge all tombstones by rehashing a block (e.g., when there are more tombstones than entries present); this operation amortizes over many deletion operations and does not affect the overall running time complexity.

Reallocation

If $\deg(u) = \beta(u)$, i. e. the adjacency block of vertex u does not have any unused indices left, we need to allocate a new adjacency block for u before we can insert new neighbors. Note that allocating a new adjacency block does not necessarily trigger an OS-level allocation (e. g. `malloc`) if our custom memory allocation scheme is used (which is described in detail in Section A of our appendix). Since the performance of our hash index depends on the fill factor of $H(u)$, it is not advisable to wait until $\deg(u) = \beta(u)$, i. e. until the fill factor

Table 1 Asymptotic complexity (amortized and in expectation) of various common data structures that implement the graph abstract data type. DHB is at least as fast as the best competing algorithm for all operations. d: degree of modified source vertex, β: size of adjacency storage ($\beta \geq d$), $n = |V|$. For simplicity, block sizes (which only yield constant speedups) are omitted from this table. We also remark that low degree vertices (e. g. the vertices which are not hashed in DHB) do not affect the overall complexities.

	Insert	Delete	Change weight	Iterate over $N(\cdot)$	Query edge	Arbitrary order
DHB	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(d)$	$\mathcal{O}(1)$	yes
Adj. arrays (e. g. STINGER)	$\mathcal{O}(d)$	$\mathcal{O}(d)$	$\mathcal{O}(d)$	$\mathcal{O}(d)$	$\mathcal{O}(d)$	yes
Adj. arrays (sorted)	$\mathcal{O}(d)$	$\mathcal{O}(d)$	$\mathcal{O}(\log d)$	$\mathcal{O}(d)$	$\mathcal{O}(\log d)$	no
Hashing only	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(\beta)$	$\mathcal{O}(1)$	no
ASPEN	$\mathcal{O}(\log n + \log d)$	$\mathcal{O}(\log n + \log d)$	$\mathcal{O}(\log n + \log d)$	$\mathcal{O}(\log n + d)$	$\mathcal{O}(\log n + \log d)$	no
TERRACE	$\mathcal{O}(\log d)$	$\mathcal{O}(\log d)$	$\mathcal{O}(\log d)$	$\mathcal{O}(d)$	$\mathcal{O}(\log d)$	no

reaches 100 %. Instead, we already reallocate the adjacency block once $\deg(u) \geq C \cdot \beta(u)$ for some constant $C < 1$ (e. g. $C = \frac{1}{2}$). When reallocating the adjacency block, we allocate a new adjacency block of block size $2 \cdot \beta(u)$, thereby increasing $\beta(u)$ to the next power of two.[3] Afterwards, we copy $N(u)$ into the new block, rebuild the hash index, update the pointers to $N(u)$ and $H(u)$ and deallocate the old adjacency block. Rebuilding the hash index is done by resetting all entries of $H(u)$ to $\square$, followed by a re-insertion of all neighbors of $N(u)$ into the hash index.

To perform actual allocations and/or deallocations, DHB can either use the system allocator (i. e. `malloc`), or a custom memory allocation scheme that is optimized for the block sizes that DHB uses. Our custom memory management works similarly to allocators in other block-based graph data structure (e. g. HORNET [5]); due to space constraints, we describe it in Section A of our appendix. In our experiments, we always use our custom memory allocator for DHB.

4.2 Operations

We briefly review the operations that DHB supports and their computational efficiency. The asymptotic running times given below are amortized over many updates (to account for reallocations) and in expectation (because of the hash index). We summarize these complexities in Table 1. In all cases, our running times are equally fast, or faster, than our competitors. Compared to traditional adjacency arrays, our hash index does not add asymptotic complexity for any operation.

Insertion of a new edge (u, v) at the end of the adjacency block first uses the hash index to check whether v is already a neighbor of u. If that is not the case, v is inserted at index $\deg(u)$ of $A(u)$ and $\deg(u)$ is incremented. This operation runs in $\mathcal{O}(1)$ time. Insertion in an arbitrary position needs to move trailing entries of $N(u)$ to free space for the new edge. Afterwards, the hash table needs to be updated for all entries that were moved. Overall, the operations runs in $\mathcal{O}(\deg(u))$.

Weight changes (or changes of other associated data) of an edge (u, v) can be performed in $\mathcal{O}(1)$ time by using the hash index to find the index of v in $N(u)$, followed by an update of the edge weight (which is stored interleaved with $N(u)$).

[3] Due to this reallocation strategy, the size of $N(u)$ can only ever reach $C \cdot \beta(u)$. Hence, it is actually enough to only allocate $C \cdot \beta(u)$ slots (and not $\beta(u)$ slots) for $N(u)$.

Deletion of an edge (u, v) first looks up the index of v in $N(u)$ by using the hash index. If the order does not need to be preserved, v can be swapped to the end of $N(u)$ and deleted in $\mathcal{O}(1)$ time. Otherwise, trailing entries of $N(u)$ need to be moved, incurring $\mathcal{O}(\deg(u))$ time.

Iteration over all neighbors of u simply iterates over the first $\deg(u)$ indices of $N(u)$, without involving $H(u)$ at all.

Edge queries check whether an edge (u, v) exists in $\mathcal{O}(1)$ time by looking up v in the hash index of u.

Reordering the neighbors of u (e. g. when sorting edges) is done by reordering $N(u)$ first. Afterwards, $H(u)$ is rebuilt from scratch in $\mathcal{O}(\deg(u))$ time.

Parallel updates

While not the focus of our data structure, we can support parallel batch updates of edges due to the fact that DHB allows adjacency blocks of distinct vertices to be manipulated concurrently. We achieve this by distributing the batch to all available threads in such a way that no threads t and t' receive edges (u, v) and (u, v') that share the same source vertex u. We achieve this by using a hash function to map source vertices u to threads. Integer sorting is used to sort the source vertices according to their hash value; afterwards, each thread applies all updates that concern the source vertices that are mapped to itself.

5 Evaluation

We perform experiments to evaluate the behavior of our data structure on several static and temporal graphs coming from SNAP [15], NR [22], and the KONECT [12] interactive data repository (see Tables 2 and 3 in Section B of the appendix). The largest graph has around 1.8B edges. Temporal graphs represent real dynamic applications and typically consist of a sequence of edges along with their timestamps. That sequence expresses some predefined pattern of edge additions related to the underlying application. We compare DHB to both static and dynamic graph frameworks. We choose NetworKit as the representative for static graph frameworks. NetworKit uses adjacency arrays to store the input graph, and the CSR representation for matrix based operations. For the comparison with dynamic graph frameworks, we choose STINGER, aspen and Terrace. STINGER is the representative framework for block-based adjacencies, aspen represents tree-based graph structures and Terrace uses different data structures depending on the neighborhood degree. We do not compare against a hashing-only implementation, as hashing alone is not competitive with other approaches when downstream algorithm performance (e. g. traversal of the data structure) is considered (and thus, state-of-the art dynamic graph data structures do not rely only on hashing).

We group the experiments into two categories. In Section 5.1, we evaluate all competitors in terms of common dynamic operations, i. e. insertions, deletions and edge weight updates. Then, in Section 5.2, we evaluate the performance of DHB for common graph applications, under dynamic and static settings. More experiments can be found in the appendix regarding: memory consumption and batch size evaluation (See Section C of the appendix) and scalability experiments for DHB (See Section D of our appendix). Experiments were conducted on a shared-memory parallel machine equipped with an 2x 18-Core Intel Xeon 6154 CPU

(2 sockets, 18 cores each), and a total of 1,5 TB RAM. [4] To ensure reproducibility, all experiments were managed by SimexPal [2]. Our code and the experimental pipeline is publicly available at `https://github.com/hu-macsy/dhb`.

Configuration of competitors

STINGER reserves half of the available physical memory and also requires the user to set the number of adjacency blocks allocated for the data structure. We set the number of expected neighbors per vector to STINGER's default (i. e. `STINGER_DEFAULT_NEB_FACTOR` $\cdot |V|$) and let STINGER use enough memory to fit these blocks into memory (i. e. 768 GiB). We do not use STINGER's client-server architecture or vertex mappings features (and also do not remap vertex IDs for all other data structures). All edge updates are performed by using STINGER's `stinger_update_directed_edge()` function.

ASPEN implements a single-writer, multi-reader interface following a lock-free approach i. e. allowing any number of concurrent readers and a single writer on a graph snapshot. To enable parallelism, ASPEN uses its own scheduler, similar to Cilk [14]. All edge insertions [deletions] are performed via ASPEN 's `insert_edges_batch()` [`delete_edges_batch()`].

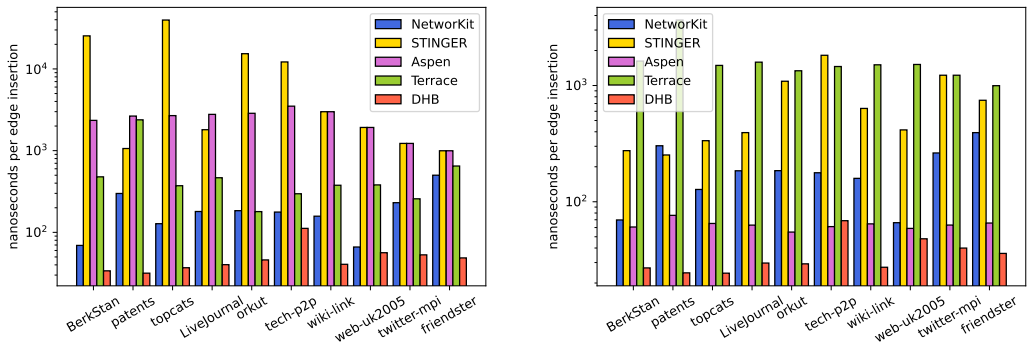
The recommended instructions for building TERRACE require a non-standard branch of the LLVM compiler (TAPIR) and use Cilk Plus [6] for multi-threading. These options introduce additional optimizations and make TERRACE difficult to compare to other, arbitrary graph libraries. More precisely, the TAPIR branch improves upon mainline LLVM by optimizing across parallel regions [23], which contributes to an increased performance for TERRACE [20]. To ensure similar build settings for all involved frameworks, we compile TERRACE with GCC and use OpenMP for multi-threading (since Cilk Plus support was recently deprecated and removed from GCC). Finally, we set TERRACE's `MEDIUM_DEGREE` to 2^{10} to avoid memory issues (after discussion with the authors of TERRACE).

5.1 Insertion, Update and Deletion Performance

For edge insertions, we perform experiments with initially empty graphs and insert all edges after verifying existence in the graph. We use two different modes for the insertion, a single-edge insertion and a bulk insertion in one batch – depicted in Figures 2a and 2b, respectively. The above experiments are performed in a single-threaded environment and include all competitions. Note that since TERRACE pre-allocates data structures in its constructor (i. e. enough memory for up to 15 edges per vertex), we also include the construction time in our measurements. This does not hinder the fairness of the experiment since all other frameworks have insignificant construction times (less than a millisecond). It is clear that DHB outperforms all competitors for both insertion modes in the single-threaded case. More precisely, DHB is on average 3.6 × faster than NETWORKIT and 9.4 × faster than TERRACE (the best competitors) for single insertions and 1.9 × faster than ASPEN (the best competitor) for the bulk insertion. Compared to STINGER, DHB is on average 17.3 [93.8] × faster for single [bulk] insertions.

Moreover, we perform experiments in a multi-threaded environment with 18 threads, depicted in Figure 3. NETWORKIT is not included in the multi-threaded experiment as it does not support parallelism. Moreover, it seems that building TERRACE with OpenMP highly penalizes its performance (causing it to run slower than in a single-threaded run).

[4] `https://www2.hu-berlin.de/macsy/technical-overview.html`



(a) Single edge insertions (one-by-one). **(b)** Bulk edge insertion (all in one batch).

Figure 2 Edge insertion experiments for static graphs on single-threaded environment.

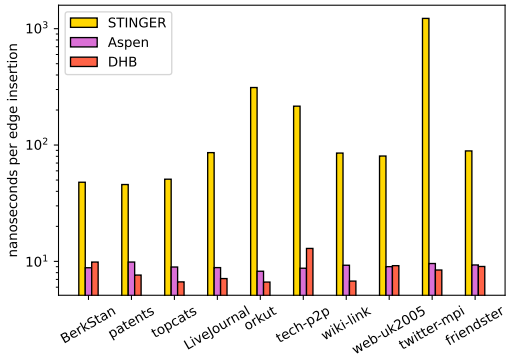
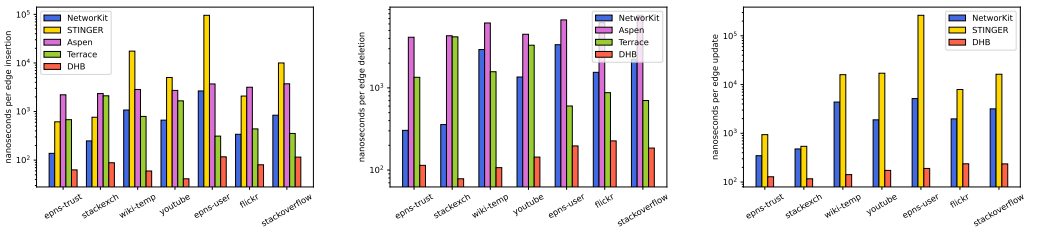


Figure 3 Bulk edge insertion for static graphs on multi-threaded environment (18 threads).

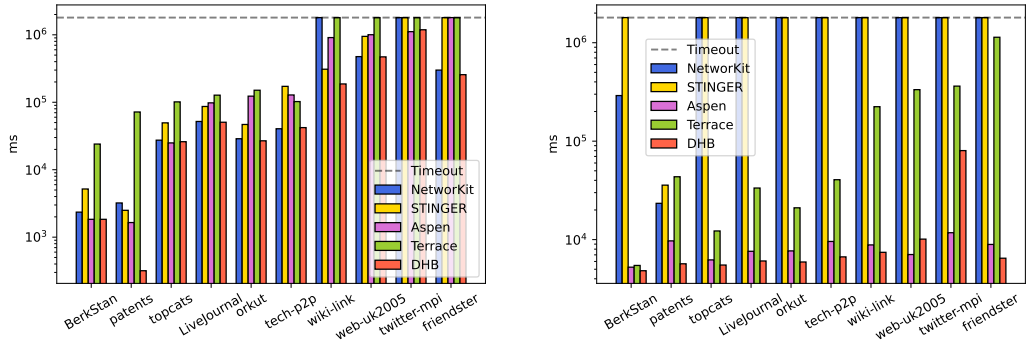
The results we obtain do not reflect the expected performance of TERRACE, as reported in the original paper [20] (which uses the custom TAPIR branch of LLVM). Therefore, to avoid ill-founded conclusions regarding TERRACE's performance in the multi-threaded experiment, we choose to exclude such results. In Figure 3, we observe that DHB is slightly faster than ASPEN (10% on average) regarding multi-threaded insertions and both competitors are on average around 14.2 × faster than STINGER.

Moreover, we perform experiments for the temporal graphs of Table 3 including edge insertions, deletions and edge weight updates (changing the weight of an edge – not inserting a new one). For deletions and weight updates, we pick the affected edges uniformly at random from the set of all edges after insertion. In this way, we do not risk altering the degree distribution of the updated graph. For edge weight updates ASPEN and TERRACE are excluded from the experiments. The former because it does not support weighted graphs while the latter because it does not offer an explicit method for edge weight updates (other than deleting and re-inserting the edge). The results are reported in Figure 4. For insertions [deletions] to temporal graphs, DHB is on average 7.4 [8.9] × faster than NETWORKIT, as seen in Figures 4a and 4b. Unfortunately, STINGER times out for all edge deletion experiments at 1 800 secs, so we exclude it from Figure 4b. The general trend is similar for edge weight updates too: DHB is 10.2 × faster than NETWORKIT and 53.1 × faster than STINGER, as seen in Figure 4c.



(a) Random edge insertions. **(b)** Random edge deletions. **(c)** Random edge weight update.

Figure 4 Single-threaded edge insertions, deletions and weight updates for temporal graphs.



(a) BFS. **(b)** SpGEMM.

Figure 5 Performance of common graph applications on top of dynamic graph structures (dynamic setting).

5.2 Applications

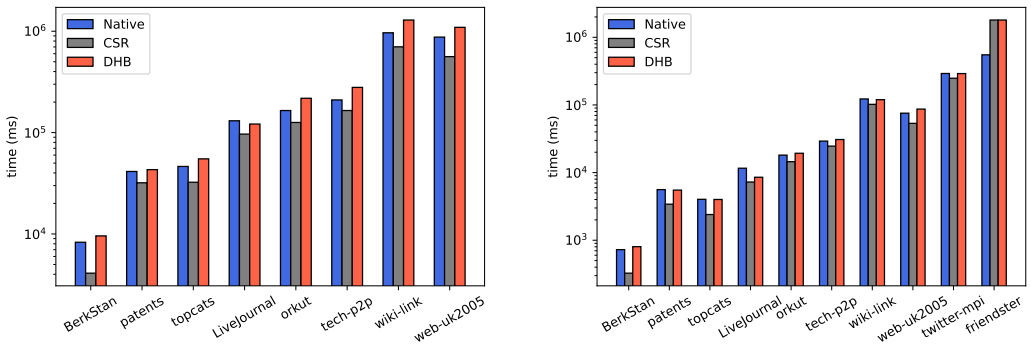
We evaluate how DHB performs in various application scenarios. For this purpose, we pick the popular BFS and SpGEMM benchmarks. We also demonstrate that DHB can easily be integrated into existing graph applications.

Breadth-first search (BFS)

We compare the performance of alternating edge insertions and BFS queries of DHB and its competitors. In this experiment, we initialize the graph to all but 10M edges (without measuring the initialization time). Afterwards, we insert 100k edges into the graph and run a BFS from a random source vertex. This process is iterated 100 times, i. e. until all edges of the graph are inserted. Identical source vertices are picked for all competitors. Figure 5a depicts the results (reporting end-to-end running time). DHB is on average 1.7 × faster than NetworKit and 2.5 × faster than Aspen.

Sparse matrix-matrix multiplication (SpGEMM)

In a second experiment, we aggregate the results of a SpGEMM computation into an existing matrix. The need to aggregate the result of a SpGEMM computation arises in various graph mining applications and/or in distributed matrix multiplication algorithms [9]. In particular, we compute the first 100M non-zeros of $\mathbf{A}^2$ where $\mathbf{A}$ is the adjacency matrix of each graph and measure the running time of this computation. We use the standard sparse row-by-row algorithm by Gustavson [10]. Note that Aspen does not support weights, but only aggregates



(a) BFS. **(b)** SpMV.

Figure 6 Comparison of DHB and NetworKit's native graph structures on static graph algorithms (no updates).

the structure and not the actual values in this experiment. Furthermore, since ASPEN only supports batch updates, we always buffer one row of the output before inserting it into ASPEN's data structure; this strategy improves ASPEN's performance considerably. All other data structures support weights and perform individual edge updates without buffering. Figure 5b shows that DHB and ASPEN report the best performance results for SpGEMM and are on average 8.4 × faster than TERRACE. Unfortunately, most of the runs time out for STINGER and NetworKit at 1 800 secs.

Integrating DHB into custom graph structures

Our final experiment demonstrates the viability of DHB as a faster drop-in replacement for custom graph data structures. We integrate DHB into the well-known graph framework NetworKit. NetworKit includes two graph data structures: a native adjacency array, and a CSR representation. We evaluate the overhead of DHB's integration compared to both NetworKit's representations. For the evaluation we pick the BFS and SpMV (= sparse matrix times dense vector multiplication) benchmarks, since they are both used as primitives in more sophisticated graph algorithms. Adjusting NetworKit's BFS and SpMV to work on top of DHB is easy to do since our data structure has the same interface as custom graph data structures. Experiments demonstrate that the DHB-enhanced version of NetworKit slightly penalizes the performance of BFS and SpMV (being on average 15% [25%] slower than NetworKit's native adjacency array [CSR] representation). The results suggest only a small overhead for graph algorithm performance compared to a significant performance improvement for edge updates. Specifically, the overhead is in line with what other authors have observed when moving from static to dynamic graphs [7].

6 Conclusions

In this work, we present DHB, a new data structure for storing and processing dynamic, large-scale, sparse graphs and matrices. DHB is designed for general-purpose CPU architectures and combines an efficient block-based memory layout to store incident edges with an additional hash index for high degree vertices. Our dynamic data structure supports edge insertions, deletions and edge weight updates. We demonstrate experimentally that DHB outperforms competing dynamic graph data structures in terms of update rates and graph applications for

both static and temporal (real) graph data. To show the viability of our data structure, we integrate DHB as a drop-in replacement for NetworKit's native dynamic graph structure (adjacency arrays). Experiments demonstrate that using DHB instead of NetworKit's native graph layout incur a small overhead for graph algorithms, while significantly increasing the update rates for edge insertions, deletions and, weight updates.

References

1. Khaled Ammar. Techniques and systems for large dynamic graphs. In Eduard C. Dragut and Heng Tao Shen, editors, *Proceedings of the SIGMOD 2016 PhD Symposium, San Francisco, California, USA, June 26, 2016*, pages 7–11. ACM, 2016.
2. Eugenio Angriman, Alexander van der Grinten, Moritz von Looz, Henning Meyerhenke, Martin Nöllenburg, Maria Predari, and Charilaos Tzovas. Guidelines for experimental algorithmics: A case study in network analysis. *Algorithms*, 12(7):127, 2019.
3. Michael A. Bender and Haodong Hu. An adaptive packed-memory array. In *Proceedings of the Twenty-Fifth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '06, pages 20–29, New York, NY, USA, 2006. Association for Computing Machinery. doi:10.1145/1142351.1142355.
4. Maciej Besta, Marc Fischer, Vasiliki Kalavri, Michael Kapralov, and Torsten Hoefler. Practice of streaming processing of dynamic graphs: concepts, models, and systems, 2021. URL: https://open.bu.edu/handle/2144/42895.
5. Federico Busato, Oded Green, Nicola Bombieri, and David A. Bader. Hornet: An efficient data structure for dynamic sparse graphs and matrices on gpus. In *The 22nd Annual IEEE High Performance Extreme Computing Conference, HPEC 2018, Waltham, MA, USA, September 25-27, 2018*, pages 1–7, Los Alamitos, CA, 2018. IEEE Computer Society. doi:10.1109/HPEC.2018.8547541.
6. Intel Corporation. Intel cilk plus language specification, 2010. URL: http://software.intel.com/sites/products/cilkplus/cilk_plus_language_specification.pdf.
7. Laxman Dhulipala, Guy E. Blelloch, and Julian Shun. Low-latency graph streaming using compressed purely-functional trees. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, pages 918–934, New York, NY, USA, 2019. Association for Computing Machinery. doi:10.1145/3314221.3314598.
8. David Ediger, Robert McColl, Jason E. Riedy, and A. David Bader. Stinger: High performance data structure for streaming graphs. *HPEC*, pages 1–5, 2012.
9. Jianhua Gao, Weixing Ji, Zhaonian Tan, and Yueyan Zhao. A systematic survey of general sparse matrix-matrix multiplication. *CoRR*, abs/2002.11273, 2020. arXiv:2002.11273.
10. Fred G. Gustavson. Two fast algorithms for sparse matrices: Multiplication and permuted transposition. *ACM Trans. Math. Softw.*, 4(3):250–269, September 1978. doi:10.1145/355791.355796.
11. Jeremy Kepner, Peter Aaltonen, A. David Bader, Aydin Buluç, Franz Franchetti, R. John Gilbert, Dylan Hutchison, Manoj Kumar, Andrew Lumsdaine, Henning Meyerhenke, Scott McMillan, E. José Moreira, D. John Owens, Carl Yang, Marcin Zalewski, and G. Timothy Mattson. Mathematical foundations of the graphblas. *HPEC*, pages 1–9, 2016.
12. Jérôme Kunegis. Konect: The koblenz network collection. In *Proceedings of the 22nd International Conference on World Wide Web*, WWW '13 Companion, pages 1343–1350, New York, NY, USA, 2013. Association for Computing Machinery. doi:10.1145/2487788.2488173.
13. Daan Leijen, Benjamin Zorn, and Leonardo de Moura. Mimalloc: Free list sharding in action. In *APLAS*, volume 11893 of *Lecture Notes in Computer Science*, pages 244–265. Springer, 2019.
14. Charles E. Leiserson. *Cilk*, pages 273–288. Springer US, Boston, MA, 2011. doi:10.1007/978-0-387-09766-4_289.

15 J. Leskovec. Stanford Network Analysis Package (SNAP). URL: http://snap.stanford.edu/index.html.

16 Chun Liu, Shuhang Zhang, Hangbin Wu, and Qiang Fu. A dynamic spatiotemporal analysis model for traffic incident influence prediction on urban road networks. *ISPRS International Journal of Geo-Information*, 6(11), 2017. URL: https://www.mdpi.com/2220-9964/6/11/362.

17 Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph Hellerstein. Graphlab: A new framework for parallel machine learning. In *Proceedings of the Twenty-Sixth Conference on Uncertainty in Artificial Intelligence*, UAI'10, pages 340–349, Arlington, Virginia, USA, 2010. AUAI Press.

18 Fredrik Manne and Mahantesh Halappanavar. New effective multithreaded matching algorithms. In *IPDPS*, pages 519–528. IEEE Computer Society, 2014.

19 Mugilan Mariappan and Keval Vora. Graphbolt: Dependency-driven synchronous processing of streaming graphs. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19, New York, NY, USA, 2019. Association for Computing Machinery.

20 Prashant Pandey, Brian Wheatman, Helen Xu, and Aydin Buluc. Terrace: A hierarchical graph container for skewed dynamic graphs. In *Proceedings of the 2021 International Conference on Management of Data*, SIGMOD/PODS '21, pages 1372–1385, New York, NY, USA, 2021. Association for Computing Machinery.

21 J. M. Robson. Worst case fragmentation of first fit and best fit storage allocation strategies. *Comput. J.*, 20(3):242–244, 1977.

22 Ryan A. Rossi and Nesreen K. Ahmed. The network data repository with interactive graph analytics and visualization. In *AAAI*, 2015. URL: http://networkrepository.com.

23 Tao B. Schardl, William S. Moses, and Charles E. Leiserson. Tapir: Embedding fork-join parallelism into llvm's intermediate representation. In *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '17, pages 249–265, New York, NY, USA, 2017. Association for Computing Machinery. doi:10.1145/3018743.3018758.

24 Dipanjan Sengupta, Narayanan Sundaram, Xia Zhu, Theodore L. Willke, Jeffrey Young, Matthew Wolf, and Karsten Schwan. Graphin: An online high performance incremental graph processing framework. In *Proceedings of the 22nd International Conference on Euro-Par 2016: Parallel Processing - Volume 9833*, pages 319–333, 2016.

25 Mo Sha, Yuchen Li, Bingsheng He, and Kian-Lee Tan. Accelerating dynamic graph analytics on gpus. *Proc. VLDB Endow.*, 11(1):107–120, September 2017.

26 Julian Shun and E. Guy Blelloch. Ligra: a lightweight graph processing framework for shared memory. *PPOPP*, pages 135–146, 2013.

27 Christian L. Staudt, Aleksejs Sazonovs, and Henning Meyerhenke. Networkit: A tool suite for large-scale complex network analysis. *Network Science*, 4(4):508–530, 2016. doi:10.1017/nws.2016.20.

28 Brian Wheatman and Helen Xu. Packed compressed sparse row: A dynamic graph representation. In *2018 IEEE High Performance Extreme Computing Conference, HPEC 2018, Waltham, MA, USA, September 25-27, 2018*, pages 1–7, September 2018. doi:10.1109/HPEC.2018.8547566.

29 Brian Wheatman and Helen Xu. A parallel packed memory array to store dynamic graphs. In Martin Farach-Colton and Sabine Storandt, editors, *Proceedings of the Symposium on Algorithm Engineering and Experiments, ALENEX 2021, Virtual Conference, January 10-11, 2021*, pages 31–45. SIAM, 2021.

30 Martin Winter, Daniel Mlakar, Rhaleb Zayer, Hans-Peter Seidel, and Markus Steinberger. Faimgraph: High performance management of fully-dynamic graphs under tight memory constraints on the gpu. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, SC '18. IEEE Press, 2018.

31 Martin Winter, Rhaleb Zayer, and Markus Steinberger. Autonomous, independent management of dynamic graphs on gpus. In *2017 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–8, 2017. doi:10.1109/HPEC.2017.8091058.

A Memory Management

Since expanding the storage associated with a vertex (i. e. increasing $\beta(u)$) happens frequently when using the DHB data structure, care must be taken to avoid costly OS-level memory allocations whenever possible. Like other block-based dynamic graph data structure, we implement a custom memory manager to allocate adjacency blocks for this purpose. While sophisticated `malloc()` implementations exist, our allocator is able to improve upon invoking `malloc()` directly because of two reasons: first, due to the design of our data structure (and the hash index in particular), we only need to manage blocks of size 2^k for some k. In contrast, general purpose `malloc()` implementations usually maintain more fine-grained size classes to avoid wasting memory on arbitrary workloads. Secondly, our block deallocation procedure does not need to recover a pointer to the (more coarse-grained) OS-level memory allocation from a pointer to the adjacency block; instead, we can store additional meta data in our data structure.

The auxiliary data structures of our memory allocator are depicted in Figure 7. As all blocked-based dynamic graph data structures, our allocation scheme groups multiple adjacency blocks into a single contiguous superblock. Hence, we only need to perform an OS-level memory allocation per superblock. In our design, all blocks of the same superblock have the same block size (i. e. the same $\beta(u)$). To protect the memory allocator against concurrent access, we use a mutex per size class (i. e. per power of two).

Block allocation

We usually use a first fit allocation strategy to allocate blocks. This allocation strategy is known to result in low fragmentation [21]. Since it causes long-lived allocations to accumulate in the first few superblocks, we expect it to reduce the number of partially occupied superblocks that we have to maintain. More precisely, we allocate from the oldest superblock of a given block size that currently has unused blocks available. For this purpose, we store all superblocks of the same block size that are not fully occupied in a balanced binary search tree ordered by their age, i. e. the order in which they were created.

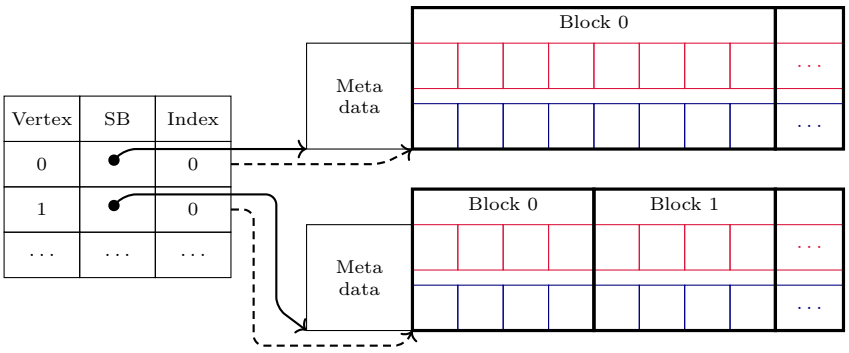


Figure 7 Layout of superblocks and block handles. Each vertex stores a pointer to its superblock and the index of its adjacency block within this superblock. The adjacency block contains both $N(\cdot)$ (red) and $H(\cdot)$ (blue); colors match Figure 1.

Table 2 Static graphs. The columns of the table correspond (in order) to network name, abbreviation, minimum degree, maximum degree, mean degree, maximum vertex ID, and edge count.

Network	Abbrev.	$\deg_{\min}$	$\deg_{\max}$	$\deg_{\text{mean}}$	$\lvert V\rvert$	$\lvert E\rvert$
web-BerkStan	BerkStan	0	249	11.1	685K	7.6M
cit-Patents	patents	0	770	2.8	6.01M	16.5M
wiki-topcats	topcats	0	3.91K	16.0	1.79M	28.5M
soc-LiveJournal1	LiveJournal	0	20.3K	14.3	4.85M	69M
com-orkut	orkut	0	33K	38.2	3.07M	117M
tech-p2p	tech-p2p	1	10.7K	25.6	5.79M	148M
web-wikipedia_link_en13	wiki-link	0	37K	20.2	27.2M	601M
web-uk-2005	web-uk2005	1	1.78M	23.7	39.5M	936M
soc-twitter-mpi-sws	twitter-mpi	0	3M	35.3	41.7M	1.47B
com-friendster	friendster	0	3.62K	14.5	125M	1.81B

Table 3 Temporal graphs from real dynamic applications. The columns of the table correspond (in order) to network name, abbreviation, minimum degree, maximum degree, mean degree, maximum vertex ID, edge count during initial read, edge cout after update accordingly.

Network	Abbrev.	$\deg_{\min}$	$\deg_{\max}$	$\deg_{\text{mean}}$	$\lvert V\rvert$	E_{in}	E_{out}
soc-epinions-trust-dir	epns-trust	0	2.07K	6.4	132K	841K	841K
ia-stackexch-user-marks-post-und	stackexch	0	4.92K	2.4	545K	1.3M	1.3M
wiki-talk-temporal	wiki-temp	0	142K	3.0	1.14M	7.83M	3.31M
soc-youtube-growth	youtube	0	83.3K	3.0	3.22M	12.2M	9.38M
rec-epinions-user-ratings	epns-user	0	162K	18.1	756K	13.7M	13.7M
soc-flickr-growth	flickr	0	26.4K	14.4	2.3M	33.1M	33.1M
sx-stackoverflow	stackoverflow	0	42.2K	10.9	2.58M	47.9M	28.2M

The are two exceptions to this first fit rule: within a superblock, we simply allocate an arbitrary block (by storing a per-superblock stack of free blocks). This does not affect fragmentation since we can only release entire superblocks to the OS at a time. Secondly, to avoid frequent modifications of the balanced binary search tree, we do not immediate re-insert a fully occupied superblock into the tree once one of its blocks is deallocated. Instead, we employ a strategy similar to the one used by mimalloc [13]. In particular, we wait until $\frac{\beta}{2}$ of a superblock's blocks are deallocated before allocating from the superblock again. This reduces the overhead of the memory allocator without affecting its asymptotic properties.

To be able to free adjacency blocks, we let each vertex store a pointer to the superblock of its current adjacency block, and the index of this adjacency block within the superblock (depicts as arrows in Figure 7. When freeing the adjacency block, we simply push its index back to the stack of free blocks that is stored within the superblock. Since block reallocations happen only $\mathcal{O}(\log k)$-times for k edge updates, our implementation does not store these information within our main per-vertex array (i. e. the array depicted in Figure 1). Instead, we use a differnet array to improve memory locality.

B Instances

All graph used in the experiments can be found in Tables 2 (static) and 3 (temporal).

C Memory and Batch Size Experiments

We perform additional experiments regarding memory consumption and scaling of the batch size.

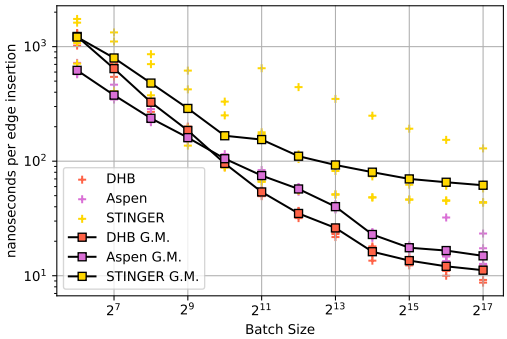


Figure 8 Edge insertion rate over the largest static graphs for an increasing batch size. We report individual and aggregated results using geometric mean over the graphs.

Batch size experiment

We use the five largest static graphs of Table 2 and scale the batch size for batched edge insertions in a parallel environment with 36 threads. Batch sizes range from 2 to 2^{17} updates per batch. The edges to be inserted are randomly generated and their existence is verified prior to insertion. In Figure 8 we report times per edge insertion for DHB, ASPEN, and STINGER (in logarithmic scale). DHB performs better than ASPEN for larger batch sizes while both exhibit a linear scaling. The best performance for DHB corresponds to a batch size of 2^{16} edges.

Memory consumption

Figure 9 presents memory utilization results for all involved data structures. We measure the peak memory consumption of each competitor for all temporal instances of Table 3. More precisely, we consider the peak resident set size after constructing and reading in the temporal graphs. DHB allocates similar amounts of memory as NETWORKIT and is on average 30% more memory-efficient that STINGER. STINGER's memory allocation is less dynamic, since it allocates multiple blocks of fixed size per neighborhood. It also does not seem to be optimized for memory efficiency. Moreover, DHB uses on average 20% less memory than ASPEN, although in some cases (see `flickr`) the trend may be opposite. ASPEN's overhead is due to the tree-based data structure which requires more space. TERRACE appears to have, on average, similar memory utilization as ASPEN. In particular, both TERRACE and ASPEN show significant memory savings over the competitors on the `epns-user` and `flickr` instances which have the highest average degrees.

D Scalability Experiments

We perform experiments to test the scalability of DHB w. r. t. a growing graph size. We also evaluate the parallel scalability of DHB in a multi-threaded parallel environment. For the first experiment, we perform $15 \times n$ random edge insertions for an increasing number of vertices, i. e. $n = 2^{20}, \ldots, 2^{26}$. In Figure 10a we observe that DHB exhibits a linear scaling behavior w. r. t. the graph size. For the parallel scalability, we perform edge insertion experiments for the five larger temporal graphs of Table 3. In Figure 10b we report aggregated speed ups of DHB w. r. t. a sequential run, for a thread count of up to 36 threads. We also include

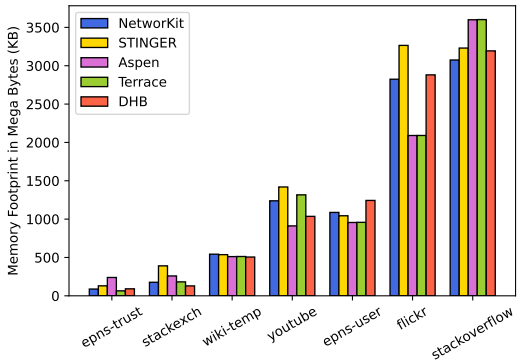
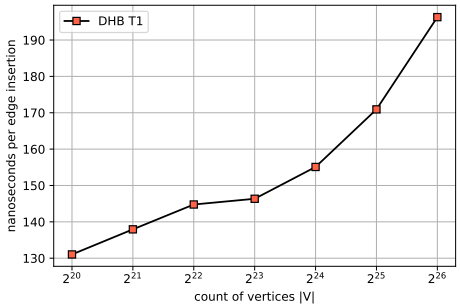
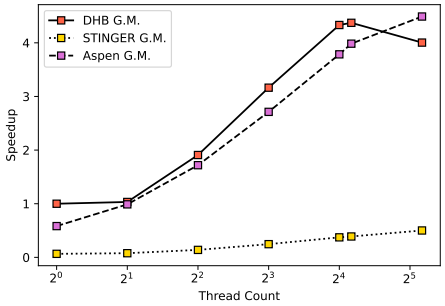


Figure 9 Memory footprint for DHB, STINGER, ASPEN, TERRACE and NETWORKIT on temporal graphs.



(a) Edge insertion rate of DHB for scaling graph size in a single-threaded execution.



(b) Geometric mean of speedups for DHB, ASPEN, STINGER on multiple threads w. r. t. a sequential run.

Figure 10 Scaling behavior of DHB w. r. t. growing number of vertices (10a) and increasing thread count (10b).

STINGER and ASPEN's speed ups for comparison. DHB scales slightly better than ASPEN while STINGER performs rather poorly. Finally, DHB's performance drops for 36 threads probably due to the NUMA issues across the two sockets of our parallel system.