Concurrent Wait-Free Graph Snapshots using Multi-Versioning

Gaurav Bhardwaj, Ayaz Ahmed, and Sathya Peri

Indian Institute of Technology Hyderabad, Hyderabad, India

Abstract. Graphs stand out as a paramount data structure for addressing real-world challenges. The binary relationships among entities or objects are vital in navigating intricate, real-time issues seen in areas like blockchain, social networks, scheduling, biological systems, and telecommunications. Unlike static graphs with immutable vertices and edges, dynamic graphs adapt to the ever-changing real-world scenarios by allowing modifications to both vertices and edges. In this context, we introduce a concurrent, lock-free dynamic graph that enables the addition, deletion, and retrieval of vertices and edges. Furthermore, we present the novel wait-free snapshot algorithm capable of both full and partial graph snapshots using multi-versioning. These snapshots pave the way for advanced graph analytics tasks, including SSSP, getpath, BFS, and more.

1 Introduction

The Graph Data Structure has emerged as a focal point of research interest within both academic and industrial circles, owing to its myriad real-life applications spanning blockchains, mapping systems, machine learning algorithms, biological networks, social networks, and more. Graphs adeptly delineate the intricate connections and structures among objects by delineating paired entity relations. Take, for instance, the use of graphs in social networks, where they serve as a visual representation of user relationships, streamlining tasks like recommendation generation, trend identification, and user behavior forecasting. Unlike traditional data structures such as linked lists, hash tables, and trees, graphs boast distinct advantages across a diverse array of application domains. Consequently, the resolution of graph-related challenges has assumed a prominent position in the realm of research across multiple disciplines.

The rise of multi-core processors has propelled parallel programming and concurrency to the forefront of research. Tailored to accommodate the escalating number of threads or tasks, concurrent data structures aim to enhance scalability, ensuring optimized performance in expansive systems or during system expansion. An array of concurrent data structures, such as Stacks[11], Queues[13,17,20], Linked-Lists [6,9,10,26,29], Hash-Tables [21,22], and others, have been developed to harness the benefits offered by multi-core processors.

Blocking mechanisms such as locks and barriers are commonly employed in concurrent applications but can introduce bottleneck issues like deadlocks. Consequently, researchers have explored non-blocking progress conditions to ensure

both efficiency and correctness. In an *obstruction-free* setting, threads operate without acquiring locks, guaranteeing that at least one thread completes its task within a finite number of steps in the absence of obstructions. *Lock-free* execution [14] ensures that at least one thread can finish its operation within a finite number of steps [14]. *Wait-free* execution [12], [14], provides the highest level of progress guarantee by ensuring that all processes can complete within a finite number of steps.

Dynamic graphs, however, present a unique challenge as they continuously evolve, with updates potentially arriving at a rapid pace, sometimes reaching tens or even hundreds of thousands of updates per second. Managing these updates concurrently in multicore environments can be particularly challenging, as synchronizing them to ensure consistent graph analytical operations becomes increasingly difficult. Real-life applications of graph analytics operations on dynamic graphs yield invaluable insights. For instance, analyzing user interactions in e-commerce or social network contexts can offer valuable insights into user behavior, potential fraud detection, and network security.

An essential criterion for concurrent data structures and algorithms is correctness. In this paper, we consider the correctness as *linearizability*[15]. A concurrent execution is linearizable if for every method in the execution, effects of the method are considered to occur instantaneously at some point denoted as *Linearization Point* (LP) between its invocation and response.

1.1 Related Work

Graph operations can be primarily categorized into two types: 1) Point Operations, which encompass adding, removing, and looking up operations, and 2) Set Operations, which involve partial or full snapshots. These snapshots can subsequently be used for more advanced operations such as BFS, getpath, SSSP, and more.

Significant advancements have recently emerged in the realm of optimizing graph data structures for multicore systems. Kallimanis et al. [16] pioneered a concurrent dynamic bounded graph, offering wait-free dynamic graph point operations and facilitating graph traversal. In a separate endeavor, Chatterjee et al. [5] introduced an unbounded concurrent linearizable graph model, supporting lock-free point operations and obstruction-free set operations. Their design leveraged a lock-free linked list to manage vertices and edges within the adjacency list. Subsequently, Chatterjee et al. [4] enhanced this structure by integrating a hash table for vertices and a linked list for edges.

Expanding upon Chatterjee et al.'s framework [5], Bhardwaj et al. [2] devised a method to create a wait-free snapshot of the graph. Their approach drew inspiration from the snapshot algorithm for iterators developed by Petrank and Timnat [24]. While this technique captures a comprehensive snapshot for executing advanced graph analytics functions, it lacks support for partial snapshots. This deficiency renders it resource-intensive for tasks such as SSSP, GetPath, BFS, and similar operations.

GraphOne [18] introduced a novel strategy for updates utilizing batch processing. They maintain versions of the adjacency list to facilitate graph analytics

operations and execute deferred updates on edges and vertices in batch form. Feng et al. [8] proposed an incremental model focused on efficiently computing graph analytical operations like Breadth-First Search and Single Source Shortest Path. Their method involves maintaining data structures optimized for these operations, which are continually updated with each modification to the graph. However, it's important to note that their implementation lacks complete dynamism, as it does not support the addition or deletion of vertices.

Multiversion concurrency control has long been employed in various domains such as database systems [28], transactional memory [19], [7], and shared memory data structures. This approach enables concurrent data updates while preserving multiple versions of the data in history, each associated with a timestamp. These distinct versions facilitate consistent data reads from different versions concurrently without interfering with ongoing update operations. Wei et al. [27] introduced a wait-free snapshot algorithm utilizing multiversioning. Similarly, Nelson et al. [23] proposed a similar approach, albeit utilizing locks for version updates. In recent years, significant research attention has been devoted to range queries and snapshot-based techniques built upon these foundational approaches.

In many contemporary applications, such as single-source shortest-path computations, reachability queries, and various approximation algorithms, there is a pressing demand for wait-free partial snapshots. These applications often require targeted access to specific portions of a graph rather than its entirety. Traditional snapshot methods, while effective for full graph access, fall short when precision and efficiency in partial graph access are paramount. Given this context, there is a clear gap in the current landscape of snapshot algorithms. To address this, we propose a wait-free snapshot(partial and full) algorithm to provide efficient, targeted access while ensuring concurrent operations remain unhindered. We have integrated the multi-version concurrency control concept from Wei et al. [**?**] into our graph implementation to achieve wait-free partial snapshots.

Our Contribution: This paper presents the design and implementation of a dynamic versioned graph data structure. Our graph data structure offers full dynamism, enabling concurrent lock-free point operations. It also incorporates the capability to perform wait-free graph set operations. The contributions of this work are summarized below:

- For a directed Graph $G = (V, E)$, we present an Abstract Data Type (ADT) in section 3.1.
- The data structure component of our graph implementation is described in section 3.2.
- Our implementation of the ADT operations is discussed in section 4.
- A comparison of our implementation with its counterparts is provided in section 5. The code is available at `https://github.com/PDCRL/VersionConcGraph.git`.

2 Preliminaries and Background

Our implementation adheres to a conventional shared memory model, where a finite set of processors is accessible by a defined number of threads operating

asynchronously. These threads interact by executing operations on shared objects and obtaining corresponding responses. The system is equipped with support for atomic read, write, and compare-and-swap (CAS) instructions.

We have devised a versioned graph data structure modeled after Chatterjee et al. [4], where vertices and edges are organized in an adjacency list format. Vertices are stored within a concurrent lock-free hash table [21], while edges are managed using a concurrent lock-free linked list [9]. To ensure consistency in graph analytics operations, we have integrated the concept of versioning from Wei et al. [**?**]. In addition to supporting graph point operations as described in [4], our implementation extends its capabilities to encompass various wait-free graph set operations, using partial and full snapshots of the graph.

Pseudocode Convention: The pseudocode featuring the Algorithm is outlined in algorithm 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 1.7, and 1.8. Throughout this paper, the pseudocode is crafted using a blend of C and JAVA-style programming languages. For accessing the member field x of a class object pointer P, we utilize the notation P.x. To convey multiple variables returned from an operation, we adopt the notation $\langle x_1, x_2, ..., x_n \rangle$.

3 Graph Data Structure

3.1 The Abstract Data Type (ADT)

We focus on implementing a simple directed weighted graph, denoted as $G = (V, E)$, where V denotes the set of vertices and E denotes the set of directed edges (ordered pairs of vertices). Each edge connects an ordered pair of vertices from V and is associated with a weight. Every vertex $v \in V$ is linked to an immutable and unique key $k \in K$, where K is a totally ordered set. A vertex $v \in V$ with key k is represented as $v(k)$. The notation $e(k, l, w)$ is used to indicate an edge from $v(k)$ to $v(l)$ in the set E with weight w.

ADT operation on G is defined below. The graph point operations are first described.

1. `AddVertex(k)` adds a vertex $v(k)$ in V if $v(k) \notin V$ and returns "`VERTEX ADDED` "otherwise returns "`VERTEX ALREADY PRESENT` ".
2. `RemoveVertex(k)` removes the vertex $v(k)$ from V if $v(k) \in V$ and returns "`VERTEX REMOVED` "otherwise returns "`VERTEX NOT PRESENT` ".
3. `AddEdge(i,j,w)` adds an edge $e(i, j, w)$ in E if $e(i, j, w) \notin E$ and returns "`EDGE ADDED` "otherwise returns "`EDGE ALREADY PRESENT` ".
4. `Remove(i, j)` removes the edge $e(i, j)$ from E if $e(i, j) \in E$, and returns "`EDGE REMOVED` "; otherwise, it returns "`EDGE NOT PRESENT` ".
5. `ContainsVertex(k)` returns "`VERTEX PRESENT` "if $v(k) \in V$ otherwise returns "`VERTEX NOT PRESENT` ".
6. `ContainsEdge(i,j)` returns "`EDGE PRESENT` "if $e(i, j, w) \in E$ otherwise returns "`EDGE NOT PRESENT` ".

We now describe the set operations implemented by us.

7. BFS (v) returns sequence of vertices in the `BFS` order if $v \in V$ otherwise returns "`VERTEX NOT PRESENT` ";

8. `SSSP (v)` returns set of all the vertices along with the distance δ from v such that $\forall u \in V \wedge$ u is connected to v $\delta(u)$ is the minimum total weight of any directed path from v to u.
9. `SNAPSHOT` returns the consistent snapshot of the graph.

While we have only delved into a subset of graph set operations within our ADT, it's important to note that our implementation is equipped to handle a wide array of such operations. The approach for these operations is akin to those we have discussed, as they rely on the partial or dynamic snapshot of the graph. This versatility allows our implementation to adapt to various graph-related tasks beyond those explicitly mentioned seamlessly.

3.2 Graph Data Structure Components

```
int global_ts

Hash H
class VNode{
  int k
  Vnode_version* vhead
}
class Vnode_version{
  int k
  int insert_ts
  int delete_ts
  Enode* elist
  Vnode_vesion* next_ver
}
```

```
class ENode{
  int l
  ENode* enext
  Enode_version* ehead
}

class Enode_version{
  int weight
  int insert_ts
  int delete_ts
  Vnode_version* dest_ver_no
  Enode_version* next_ver
}
```

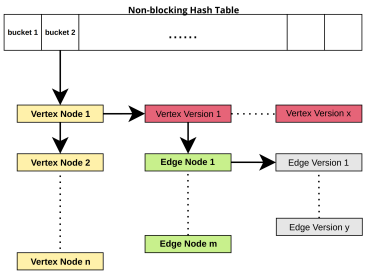

Fig. 1: Graph Datastructure Components.

In our implementation, we have maintained vertices, denoted as **`Vnode`**, within a concurrent resizable hash table H [21]. Each **`Vnode`** maintains a list of versions, documenting the history of a specific vertex through a pointer variable known as **`vhead`**. Every version of a vertex, referred to as **`Vnode_version`**, records the vertex's history by capturing both the insert and delete timestamp of that version. These timestamps facilitate the determination of the time span during which a particular vertex was active. Vnode_version node in the version lists are maintained in the descending order of their insert timestamp where pointer **`next_ver`** points to the next version available. These versions ensure that a new entry is not appended to the list unless the delete timestamp of the preceding version is established, thus preserving the sequential order.

Additionally, each **`Vnode_version`** maintains a list of all outgoing edges in the form of a linked list, with the individual edges represented as **`Enode`** nodes. Similar to **`Vnode`**, we adopt a versioning approach to track the history of edges. Each edge version list is comprised of **`Enode_version`** nodes, where each **`Enode_version`** captures both the insert timestamp and the delete timestamp of that specific version. Notably, each **`Enode_version`** also preserves a pointer to the corresponding destination vertex version node, simplifying graph traversal.

Additionally, we have a global shared counter *global_ts* to track the current timestamp of the data structure. Whenever a graph update occurs, *global_ts* is

used to record the timestamp of the update. This enables us to pinpoint precisely when a particular data element was active within the data structure. Furthermore, each time a graph set operation commences, the system atomically fetches and increments this global timestamp. This mechanism allows us to examine all vertices and edges that existed during that specific timespan. For more details, please refer to Figure 1.

4 Algorithm Description and Reasoning

This section will elaborate on the various ADT operations discussed in Section 3.1. As previously mentioned, our data structure utilizes an adjacency list where vertices are stored in a lock-free concurrent Hash Table. Each vertex is managed within a bucket in the Hash Table, and the outgoing edges are organized in the form of a lock-free linked list as shown in Figure 1.

The initiation of any ADT operation involves a search for the vertex within the hash table. The hash table comprises buckets containing vertices arranged in an unsorted linked list. When searching for a vertex, it traverses the complete list of vertices within the relevant bucket. Adding a vertex not currently present in the vertex list of its bucket is achieved atomically by placing it at the head of the linked list. This ensures a streamlined and efficient addition process. Furthermore, our approach incorporates versioning when removing a vertex from the linked list. During removal, we set the delete_ts (delete timestamp) for the vertex, applying a version control concept for effective data structure management.

```
1:  AddVertex(i)
2:    curr_Vnode ← H.FindV(i)
3:    if (curr_Vnode = nullptr) then
4:      newVnode = new VNODE(i)
5:      if (H.INSERT (newVnode) ) then
6:        setTs(newVnode)
7:        return VERTEX ADDED
8:      else
9:        return VERTEX ALREADY PRESENT
10:   else
11:     curr_ver ← READ(curr_Vnode.vhead)
12:     if (curr_ver.delete_ts = ∞) then
13:       return VERTEX ALREADY PRESENT
14:     setTs(curr_ver)
15:     new_ver = new VNODE_VERSION(i)
16:     if (cas(curr_Vnode.vhead, curr_ver, new_ver))
        then
17:       setTs(new_vesion)
18:       return VERTEX ADDED
19:     else VERTEX ALREADY PRESENT
```

Algorithm 1.1: AddVertex Operation.

4.1 Lock-free Vertex operations

AddVertex operation is given in lines 1 to 19 in Algorithm 1.1. The `FindV` method, denoted at line 2, conducts a search for the vertex in the Hash Table and returns the corresponding *VNode* if the vertex is present; otherwise, it returns *nullptr*. If the vertex is not found in the Hash Table, a new *VNode* is created at line 4, and an attempt is made to insert it into the Hash Table at line 5.

In the case where the insert operation fails, it implies that another concurrent thread has successfully inserted the *VNode* into the Hash table, and it simply returns "`VERTEX ALREADY PRESENT` "at line 9. On the other hand, if the insertion of the *VNode* into the hash table is successful, the timestamp of the newly inserted *VNode* is set at line 6. Finally, the operation concludes by returning "`VERTEX ADDED` "at line 7.

```
20: RemoveVertex(i)
21:   curr_Vnode ← H.FindV(i)
22:   if (curr_Vnode = nullptr) then
23:     return VERTEX NOT PRESENT
24:   else
25:     curr_ver ← READ(curr_Vnode.vhead)
26:     if (curr_ver.delete_ts = ∞) then
27:       if (cas(curr_ver.delete_ts, ∞, −1)) then
28:         setTs(curr_ver)
29:         return VERTEX REMOVED
30:       else
31:         setTs(curr_ver)
32:         return VERTEX NOT PRESENT
33:     else
34:       setTs(curr_ver)
35:       return VERTEX NOT PRESENT
```

Algorithm 1.2: RemoveVertex Operation.

```
36: ContainsVertex(i)
37:   curr_Vnode ← H.FindV(i)
38:   if (curr_Vnode = nullptr) then
39:     return VERTEX NOT PRESENT
40:   else
41:     curr_ver ← READ(curr_Vnode.vhead)
42:     if (curr_ver.delete_ts = ∞) then
43:       return VERTEX PRESENT
44:     else
45:       setTs(curr_ver)
46:       return VERTEX NOT PRESENT
```

Algorithm 1.3: ContainsVertex Operation.

If the vertex is already present in the Hash Table, the `FindV` method returns the corresponding *VNode*. It proceeds to read the current version from the version list of the $VNode$ associated with the vertex, located at its *vhead*. If the *delete_ts* (delete timestamp) of the current version is set to ∞, it indicates that the vertex exists, and we can return "`VERTEX ALREADY PRESENT` "at line 13.

However, if the delete timestamp suggests that the vertex has been marked for deletion, we set the timestamp of the version at line 14 (if not already set) using method setTs (given in Algorithm 1.8) and initiate the creation of a new version. By trying to set the *delete_ts*, the AddVertex is helping the RemoveVertex method. Subsequently, an attempt is made to atomically add the newly created vertex version to the head of the version list using `CAS` at line 16. Upon a successful `CAS` operation, the timestamp (*insert_ts*) for the newly added vertex version is set to the current global timestamp, and the operation returns "`VERTEX ADDED` ". In the event of a `CAS` operation failure, signifying that another concurrent thread has successfully added the version, the operation concludes by returning "`VERTEX ALREADY PRESENT` "at line 19.

```
47: AddEdge(i, j, w)
48:   ⟨sv, dv⟩ ← H.FindVplus(i, j)
49:   if (sv ∨ dv) then
50:     return VERTEX NOT PRESENT
51:   else
52:     c_ENode ← sv.FindE(j)
53:     if (c_ENode = nullptr) then
54:       n_Enode ← new Enode(j, d_ver, w)
55:       if (sv.insert(n_Enode)) then
56:         setTs(n_Enode)
57:       else
58:         c_ENode ← sv.FindE(j)
59:         goto Insert Version
60:     else
61:       Insert Version:
62:       c_Ever ← READ(c_Enode.vhead)
63:       if (c_Ever.delete_ts = ∞
              ⋀ c_Ever.weight = w) then
64:         return EDGE ALREADY PRESENT
65:       n_Ever ← new Enode_version(dv, w)
66:       if (cas(c_Enode.vhead, c_Ever, n_Ever))
      then
67:         setTs(c_Enode)
68:       else goto Insert Version
69:     if (sv.delete_ts ≠ ∞) then
70:       setTs(sv)
71:       if (sv.delete_ts < n_Ever.insert_ts)
      then
72:         return VERTEX NOT PRESENT
73:     if (dv.delete_ts ≠ ∞) then
74:       setTs(dv)
75:       if (dv.delete_ts < n_Ever.insert_ts)
      then
76:         return VERTEX NOT PRESENT
77:     return EDGE ADDED
```

Algorithm 1.4: AddEdge Operation.

RemoveVertex operation is detailed from lines 20 to 35 in Algorithm 1.2. Utilizing a multi-versioning approach for consistent snapshots, this operation refrains from physically deleting a vertex. Instead, it marks the active version as deleted by setting the delete timestamp (*delete_ts*). Following a similar pattern to AddVertex, this operation commences by locating the vertex in the hash table.

If the vertex is not present in the hash table, the operation promptly returns "`VERTEX NOT PRESENT` "at line 23. In the event that the vertex is found in the Hash Table, it examines the *curr_ver* (current version) on the version list of the corresponding $VNode$. If the *delete_ts* of the *curr_ver* is not set to ∞, indicating that the vertex exists. Setting the *delete_ts* undergoes a two-step process. Initially, it is atomically set to -1 using the `CAS` operation at line 27, signaling to other threads that this version is marked for deletion. Subsequently, the *delete_ts* is set to the current global timestamp at line 28, and the operation returns "`VERTEX REMOVED` "at line 29.

```
78:  RemoveEdge(i, j)
79:    ⟨sv, dv⟩ ← H.FindVplus(i, j)
80:    if (sv ∨ dv) then
81:      return VERTEX NOT PRESENT
82:    else
83:      c_ENode ← sv.FindE(j)
84:      if (c_ENode = nullptr) then
85:        return EDGE NOT PRESENT
86:      else
87:        c_Ever ← read(c_Enode.vhead)
88:        if (c_Ever.delete_ts ≠ ∞) then
89:          setTs(c_Enode)
90:          return EDGE NOT PRESENT
91:        else
92:          cas(c_Ever.delete_ts, ∞, −1)
93:          setTs(c_Ever)
94:      if (sv.delete_ts ≠ ∞) then
95:        setTs(sv)
96:        if (sv.delete_ts < c_Ever.delete_ts)
     then
97:          return VERTEX NOT PRESENT
98:      if (dv.delete_ts ≠ ∞) then
99:        setTs(dv)
100:       if (dv.delete_ts < c_Ever.delete_ts)
     then
101:        return VERTEX NOT PRESENT
102:    return EDGE REMOVED
```

Algorithm 1.5: RemoveEdge Operation.

If the `CAS` operation fails, implying that another concurrent thread has set the *delete_ts* to -1, the operation proceeds to set the timestamp for the current version and returns "`VERTEX NOT PRESENT` "at line 32. Similarly, if the *delete_ts* of the current version is not ∞, indicating that it is already marked for deletion, the *delete_ts* is set to the current global timestamp, and the operation returns "`VERTEX NOT PRESENT` "at line 35.

ContainsVertex operation is given in line 36 to 46. Upon invocation, it yields "`VERTEX PRESENT` "if the vertex is indeed present; conversely, it yields "`VERTEX NOT PRESENT` "if the vertex is absent.

The algorithm first seeks the corresponding $Vnode$ for the vertex within the Hash table at line 37. Should the vertex not be found in the Hash Table, indicating its absence, the algorithm promptly returns "`VERTEX NOT PRESENT` "at line 39. Conversely, if the $Vnode$ is found within the Hash Table, the algorithm proceeds to check whether the deletion timestamp (*delete_ts*) for the current version (*curr_ver*) of the vertex is set. In the event that the timestamp is not set, denoting that the vertex has not been deleted, the algorithm concludes by returning "`VERTEX PRESENT` ". Conversely, if the timestamp is set, implying the

vertex has been deleted, the algorithm updates the timestamp at line 45 and then returns "`VERTEX NOT PRESENT` ".

4.2 Lock-Free Edge operations

AddEdge is described in Algorithm 1.4, facilitates the addition of an edge between vertices i and j with weight w. It begins by inspecting the vertices in the Hash Table at line 48 using the `FindVplus` method. This method retrieves the latest vertex version for both the source and destination vertices if their delete timestamps are not set. If either vertex is not present or their deletion timestamp is already set, indicating deletion, the operation returns `nullptr` and "`VERTEX NOT PRESENT` "is returned at line 50.

If both source and destination vertices exist in the Hash Table, the algorithm checks if the $ENode$ for the destination vertex exists in the edge lists of the source vertex. The `FindE` method returns the $ENode$ for an outgoing edge to the destination vertex at line 52; otherwise, it returns `nullptr`. If no $ENode$ exists for the destination vertex, a new $ENode$ is created and atomically inserted into the edge list at line 55. Upon successful insertion, the timestamp for the newly added edge is set at line 56. If the insertion fails due to concurrent modification by another thread, the algorithm attempts to find the $ENode$ for the destination vertex again in the edge list at line 58, and proceeds to add a new edge version node on the newly found $ENode$.

Once the $Enode$ for the destination vertex is found in the edge list, we find its latest edge version. If the delete timestamp of the latest edge version is not set and the weight is also equal to the current weight, we simply return "`EDGE ALREADY PRESENT` "at line 64. Otherwise, it creates a new edge version node at line 65 and atomically adds the new version on the vhead of $Enode$ using CAS. If CAS succeeds, then it sets the timestamp at line 67. If the CAS fails, then some other thread must have added a new version and we retry adding a new version from line 61.

```
103: ContainsEdge(i, j)
104:     ⟨sv, dv⟩ ← H.FindVplus(i, j)
105:     if (sv ∨ dv) then
106:         return VERTEX NOT PRESENT
107:     else
108:         c_ENode ← sv.FindE(j)
109:         if (c_ENode = nullptr) then
110:             return EDGE NOT PRESENT
111:         else
112:             c_Ever ← read(c_Enode.vhead)
113:             if (c_Ever.delete_ts ≠ ∞) then
114:                 setTs(c_Ever)
115:                 return EDGE NOT PRESENT
116:         if (sv.delete_ts ≠ ∞) then
117:             setTs(sv)
118:             if (sv.delete_ts < c_Ever.insert_ts)
    then
119:                 return VERTEX NOT PRESENT
120:         if (dv.delete_ts ≠ ∞) then
121:             setTs(dv)
122:             if (dv.delete_ts < c_Ever.insert_ts)
    then
123:                 return VERTEX NOT PRESENT
124:         return EDGE PRESENT
```

Algorithm 1.6: ContainsEdge Operation.

Upon a successful CAS operation, it's possible that either the source or destination vertex has already been deleted prior to the insertion of the edge. In such scenarios, we verify whether the delete timestamp for the source and destination vertices is already set or not, as indicated in lines 69 and 73, respectively.

If either the source or destination vertex is already deleted, we examine whether the delete timestamp of the vertex is less than the insertion timestamp of the edge. In this situation, we can infer that the edge was added after the deletion of a vertex, and therefore, we simply return "`VERTEX NOT PRESENT` "without altering the newly added edge.

If the delete timestamp of the vertex is the same as the edge insertion timestamp, we can linearize the edge's insertion just before the vertex's deletion. Conversely, if the delete timestamp of the vertex is greater than the insert timestamp of the edge, we can conclude that the vertex was deleted after the insertion of the edge, and thus, we return "`EDGE ADDED` ". In cases where the delete timestamp of the vertex is not set, indicating the vertex still exists, the addition of the edge is deemed successful.

RemoveEdge method, is described in Algorithm 1.5, serves to eliminate the edge connecting the source vertex and the destination vertex from the graph, should it exist. Analogous to the AddEdge operation, RemoveEdge begins by searching for the source and destination vertices, returning "`VERTEX NOT PRESENT` "if either vertex is absent. Subsequently, it locates the ENode corresponding to the destination vertex in the edge list of the source vertex. If the ENode is absent, it signifies that the edge does not exist. Conversely, if the ENode is present, RemoveEdge examines the delete timestamp of the latest version. If the delete timestamp is already set, indicating that the edge has been deleted, "`EDGE NOT PRESENT` "is returned. Otherwise, the delete timestamp is initially set to -1, followed by setting it to the current global timestamp.

Following the setting of the delete timestamp of the edge, RemoveEdge verifies whether the source or destination vertex has been deleted. If either vertex has been deleted, it checks whether the deletion occurred before the edge deletion. In such a scenario, "`VERTEX NOT PRESENT` "is returned; otherwise, "`EDGE REMOVED` "is returned.

ContainsEdge method, detailed in Algorithm 1.6, is designed to determine the presence of an edge and retrieve its weight if it exists. Similar to preceding edge-related methods, it begins by verifying the presence of both the source and destination vertices. If either vertex is not found, the method promptly returns "`VERTEX NOT PRESENT` ". Subsequently, it checks the presence of the edge itself. If the edge is not found, "`EDGE NOT PRESENT` "is returned.

In the event that the edge is indeed present, ContainsEdge further ensures that neither the source vertex nor the destination vertex has been concurrently deleted before the edge was added, as indicated by checking the timestamp of the deleted vertex. If such deletion is detected, "`VERTEX NOT PRESENT` "is returned to indicate that the edge cannot be accessed due to the deletion of one of its vertices. Otherwise, the weight of the edge is returned as expected.

4.3 Wait-Free Graph Set Operation

All graph set operations, such as SSSP, BFS, Snapshot, Getpath, leverage the versioning mechanism previously discussed. Consequently, an atomic fetch and increment operation is performed on the global timestamp before initiating any graph set operation. Subsequently, the operation traverses through the

```
125: BFS(i)
126:     ts ← getTS()
127:     ⟨v_ver⟩ ← H.FindVplus(i, ts)
128:     if (v_ver = nullptr) then
129:         return VERTEX NOT PRESENT
130:     else
131:         queue⟨Vnode_version⟩ Q
132:         list bfs_l
133:         bfs_l .insert(v_ver.k)
134:         Q.insert(v_ver)
135:         while (!Q.empty()) do
136:             top ← Q.pop()
137:             c_edge ← top.enext
138:             while c_edge do
139:                 e_ver ← FindEv(c_edge, ts)
140:                 if (e_ver) then
141:                     d_v ← e_ver.dest_ver
142:                     if (d_v.k ∉ bfs_l) then
143:                         bfs_l.insert(d_v.k)
144:                         Q.push(d_v)
145:                 c_edge ← c_edge.enext
146:     return bfs_l
```

Algorithm 1.7: BFS Operation.

vertices or edges that were active during the timestamp retrieved, ensuring a consistent snapshot. Due to space constraints, we focus solely on discussing the BFS operation in this paper, as the remaining graph set operations follow a similar approach.

```
147: setTs(Vnode_version v)
148:     c_ver ← v.head
149:     if (c_ver.insert_ts = −1) then
150:         ts ← global_ts
151:         cas(c_ver.insert_ts, −1, ts)
152:     if (c_ver.delete_ts = −1) then
153:         ts ← global_ts
154:         cas(c_ver.delete_ts, −1, ts)
155: read(Vnode_version v)
156:     if (v.insert_ts = −1 ∨ v.delete_ts =
     −1) then
157:         setTs(v)
158:     return v
159: getTS()
160:     ts ← global_ts
161:     cas(global_ts, ts, ts + 1)
162:     return ts
```

Algorithm 1.8: Pseudocode of setTs, read and getTS Operation.

BFS This method systematically explores a graph's vertices reachable from a specified source vertex in BFS (Breadth-First Search) order. Commencing with the getTS operation at line 126, the algorithm initiates an atomic fetch and increment operation on the global timestamp. The retrieved timestamp, denoted as t, serves as the reference point for collecting all vertices and edges existing at that particular timestamp.

The FindVplus method returns an active version of the source vertex (if it exists) based on the timestamp t. If no vertex version is identified, it gracefully returns nullptr, signaling a non-existent vertex version. The algorithm then proceeds to create a queue that encapsulates all the vertex version nodes requiring traversal, commencing from the source vertex node. Simultaneously, a list is instantiated to track the order in which vertices are traversed during the BFS exploration.

The algorithm strategically dequeues elements one by one, inspecting all edges present at timestamp t. It appends the destination vertices of these edges to the queue and to the BFS list if they have not been visited previously. This process continues until the queue is empty, indicating the completion of the BFS traversal. Subsequently, the algorithm returns the list of vertices traversed during the BFS exploration.

4.4 Memory Management

We have leveraged the concept of multi-version concurrency control [19] to execute consistent snapshots efficiently. As the number of update operations escalates, the count of version nodes in the graph increases as well. This often results in storing versions that far are too old to serve any useful purpose, thus leading to bloating of memory. To address this challenge, we have integrated **DEBRA** [3], a lock-free memory reclamation algorithm, into our solution. Our implementation meticulously tracks ongoing graph-set operations by threads along with their timestamps. This enables us to identify outdated versions within the graph's history and reclaim those versions accordingly, thereby ensuring optimal memory utilization.

5 Evaluations

Experimental Setup. Our experimentation took place on a system equipped with an AMD EPYC 7452 CPU housing 64 cores, featuring a clock speed spectrum ranging from 1.5 GHz to 2.3 GHz. Each core operates with two logical threads and possesses exclusive 32KB L1 data and instruction caches. Collaboratively, core pairs share a 512KB L2 cache and a substantial 16MB L3 cache. The system's robust configuration extends to 252GB of RAM and a 2TB hard disk. Running on Ubuntu 18.04.6 LTS. The code, compiled using g++ 11.1.0 with -std=c++17, is intricately linked with pthread and atomic libraries. Our methodology involves averaging results from multiple runs, with a strategic cache pre-warming during the initial iterations. [a]

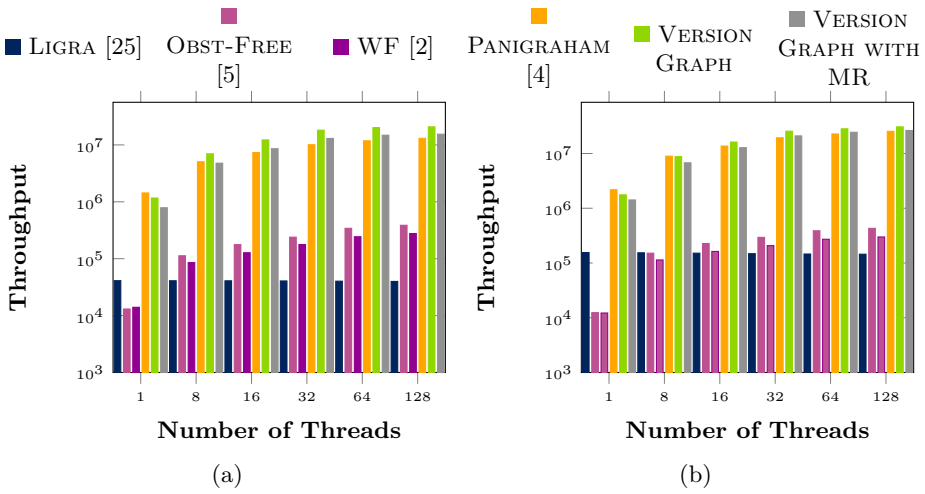

Fig. 2: Performance of our implementation compared to its counterparts for point operations. x-axis: Number of threads. y-axis: Throughput. a) Update Heavy Workload, b) Read Heavy Workload.

[a] The code is available at https://github.com/PDCRL/VersionConcGraph.git

In each experiment, we initialize the data structure with input vertices and edges of wiki-vote [1] dataset. Subsequently, we generate a random permutation for ADT point operations and graph set operations. The distribution is over the following ordered set of operations: AddVertex, RemoveVertex, ContainsVertex, AddEdge, RemoveEdge, ContainsEdge. Specifically:

1. Read Heavy Workload: 3%, 2%, 45%, 3%, 2%, 45%
2. Update Heavy Workload: 12%, 13%, 25%, 13%, 12%, 25%, 2%

For graph set operations, the distribution equally samples from all other point operations.

Algorithms: We have conducted a comparative analysis between our versioned graph algorithm and several state-of-the-art graph algorithms, including: (a) Ligra [25] (b) Obstruction Free Graph Snapshot (Obst-Free) [5], (c) Wait-free Graph Snapshot (WF) [2], (d) Panigraham [4]. We aimed to encompass all algorithms that are fully dynamic, allowing concurrent addition and removal of vertices and edges. We chose not to compare our results with GraphOne [18], as Panigraham [4] significantly outperforms it.

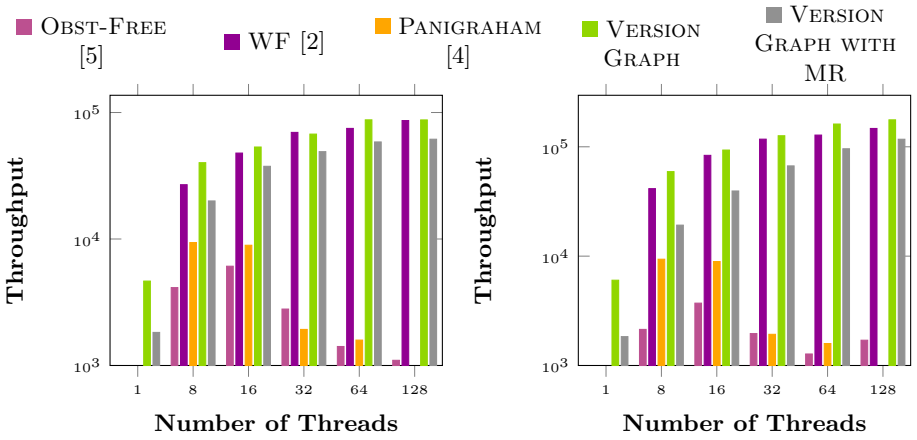

Fig. 3: Performance of our implementation compared to its counterparts for complete snapshot operation. x-axis: Number of threads. y-axis: Throughput. a) Update Heavy Workload with complete Snapshot, b) Read Heavy Workload with complete snapshot.

Memory Management imposes a substantial overhead on the performance of lock-free data structures. Interestingly, some of the algorithms we compared with did not incorporate any memory reclamation technique. Thus, in our results, we present findings for both scenarios: with and without memory reclamation.

Graph Point Operation In our comparative analysis, we assessed our implementation for point operations against state-of-the-art algorithms in Figure

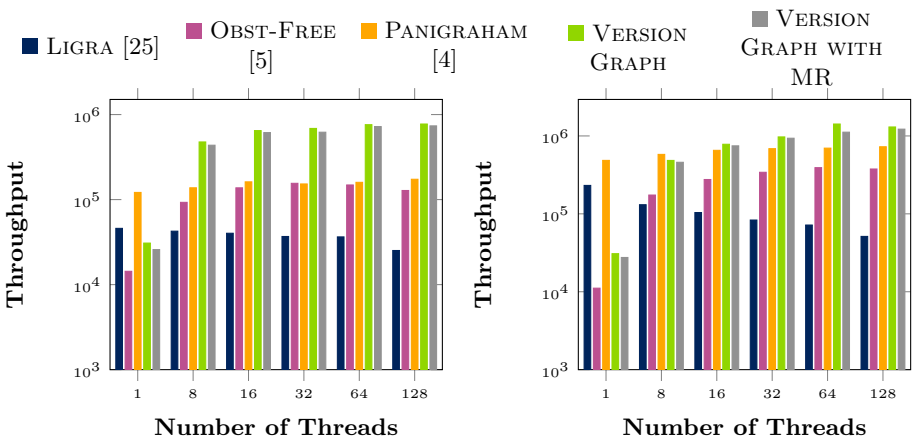

Fig. 4: Performance of our implementation compared to its counterparts for BFS operation. x-axis: Number of threads. y-axis: Throughput. a) Update Heavy Workload with BFS, b) Read Heavy Workload with BFS.

2. Despite the overhead of maintaining versions, our Version Graph algorithm demonstrated superior performance compared to its counterparts. It outperformed all other algorithms except Panigraham [4], by a large degree of magnitude. However, when compared to Panigraham [4], Version Graph showcased a noteworthy improvement, achieving a 20% higher throughput. Furthermore, our loss in throughput due to garbage collection remained minimal, especially considering the absence of concurrent graph-set operations. We conducted comparisons with and without memory reclamation, considering that many of our counterparts do not employ any memory reclamation techniques.

Graph Complete Snapshot Operation The complete snapshot operation of a graph plays a crucial role in concurrent graph processing, particularly for tasks like calculating Betweenness Centrality, All Pair Shortest Path, Diameter, etc. This operation enables efficient execution of such tasks by providing a consistent view of the graph. We have compared our implementation with its counterparts for a complete snapshot in Figure 3. Our implementation of the complete snapshot operation showcased remarkable performance compared to its obstruction-free counterparts, exhibiting a superiority with a large degree of magnitude. Notably, without memory reclamation, our multi-version graph surpassed WF [2] by a significant margin of 20%, primarily attributable to the substantial overhead of graph construction. While we incurred some performance loss due to memory reclamation, it's important to note that we compared our implementation without memory reclamation against counterparts as all its counterparts lack memory reclamation mechanism.

Graph Set Operation with partial snapshot In Figure 4 and 5, we compared our implementation against counterparts in terms of graph set operations like BFS and SSSP. Leveraging the wait-free nature of our algorithm and others being obstruction-free, we managed to surpass them in performance with a sig-

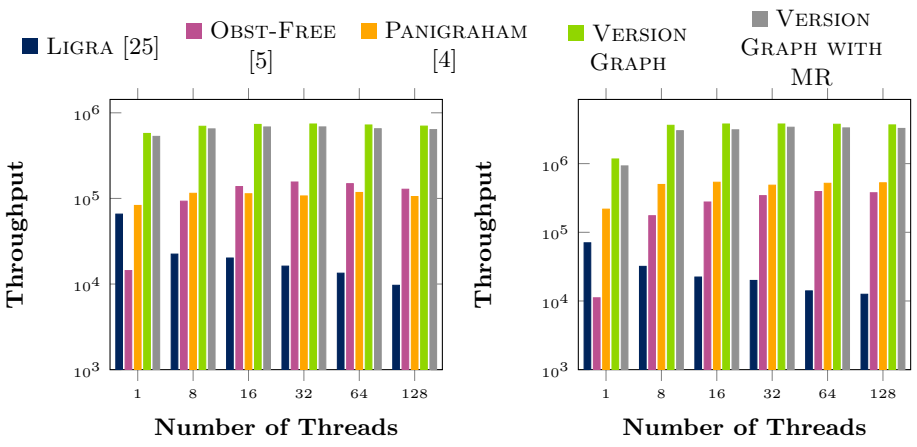

Fig. 5: Performance of our implementation compared to its counterparts for SSSP operation. x-axis: Number of threads. y-axis: Throughput. a) Update Heavy Workload with SSSP, b) Read Heavy Workload with SSSP.

nificant degree of magnitude. Notably, with a single thread, the obstruction-free variants exhibited superior performance compared to the wait-free version, owing to the absence of obstructions. However, as the number of threads increased, the performance of the obstruction-free algorithm declined due to an escalation in obstructions.

6 Conclusion

In our implementation, we introduce a novel approach to concurrent graph processing, incorporating a wait-free mechanism for both partial and complete snapshots using multi-version concurrency control. Our algorithm ensures lock-free execution for all point operations and wait-free performance for graph analytics operations, surpassing its counterparts by a significant margin. Notably, our implementation marks the first instance of a wait-free partial snapshot operation in graph processing. However, the current implementation performs point operation lock-free, so making it wait-free is another future work.

References

1. SNAP Stanford Network Analysis Project: Wiki-Vote dataset. `https://snap.stanford.edu/data/wiki-Vote.html`. Accessed: March 2024.
2. Gaurav Bhardwaj, Sathya Peri, and Pratik Shetty. Brief announcement: Non-blocking dynamic unbounded graphs with wait-free snapshot. In *International Symposium on Stabilizing, Safety, and Security of Distributed Systems*, pages 106–110. Springer, 2023.
3. Trevor Alexander Brown. Reclaiming memory for lock-free data structures: There has to be a better way. In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing*, pages 261–270, 2015.
4. Bapi Chatterjee, Sathya Peri, Muktikanta Sa, and Komma Manogna. Non-blocking dynamic unbounded graphs with worst-case amortized bounds. In *International Conference on Principles of Distributed Systems*, 2021.

5. Bapi Chatterjee, Sathya Peri, Muktikanta Sa, and Nandini Singhal. A Simple and Practical Concurrent Non-blocking Unbounded Graph with Linearizable Reachability Queries. In *ICDCN 2019, Bangalore, India, January 04-07, 2019*, pages 168–177, 2019.
6. Bapi Chatterjee, Ivan Walulya, and Philippas Tsigas. Help-optimal and Language-portable Lock-free Concurrent Data Structures. In *ICPP*, pages 360–369, 2016.
7. Ved P. Chaudhary, Chirag Juyal, Sandeep Kulkarni, Sweta Kumari, and Sathya Peri. Achieving starvation-freedom in multi-version transactional memory systems. In *Networked Systems: 7th International Conference, NETYS 2019, Marrakech, Morocco, June 19–21, 2019, Revised Selected Papers*, page 291–310, Berlin, Heidelberg, 2019. Springer-Verlag. doi:10.1007/978-3-030-31277-0_20.
8. Guanyu Feng, Zixuan Ma, Daixuan Li, Shengqi Chen, Xiaowei Zhu, Wentao Han, and Wenguang Chen. Risgraph: A real-time streaming system for evolving graphs to support sub-millisecond per-update analysis at millions ops/s. In *Proceedings of the 2021 International Conference on Management of Data*, pages 513–527, 2021.
9. Timothy L. Harris. A Pragmatic Implementation of Non-blocking Linked-Lists. In *DISC*, pages 300–314, 2001.
10. Steve Heller, Maurice Herlihy, Victor Luchangco, Mark Moir, William N. Scherer III, and Nir Shavit. A Lazy Concurrent List-Based Set Algorithm. *Parallel Processing Letters*, 17(4):411–424, 2007.
11. Danny Hendler, Nir Shavit, and Lena Yerushalmi. A scalable lock-free stack algorithm. *J. Parallel Distrib. Comput.*, 70(1):1–12, jan 2010.
12. Maurice Herlihy. Wait-free Synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, January 1991.
13. Maurice Herlihy, Victor Luchangco, and Mark Moir. Obstruction-Free Synchronization: Double-Ended Queues as an Example. In *(ICDCS*, pages 522–529, 2003.
14. Maurice Herlihy and Nir Shavit. On the Nature of Progress. In *OPODIS*, pages 313–328, 2011.
15. Maurice P Herlihy and Jeannette M Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990.
16. Nikolaos D. Kallimanis and Eleni Kanellou. Wait-Free Concurrent Graph Objects with Dynamic Traversals. In *OPODIS*, pages 1–27, 2015.
17. Alex Kogan and Erez Petrank. Wait-Free Queues With Multiple Enqueuers and Dequeuers. In *PPOPP*, pages 223–234, 2011.
18. Pradeep Kumar and H Howie Huang. Graphone: A data store for real-time analytics on evolving graphs. *ACM Transactions on Storage (TOS)*, 15(4):1–40, 2020.
19. Priyanka Kumar, Sathya Peri, and K. Vidyasankar. A TimeStamp Based Multi-version STM Algorithm. In *ICDCN*, pages 212–226, 2014.
20. Edya Ladan-Mozes and Nir Shavit. An Optimistic Approach to Lock-free FIFO Queues. *Distributed Computing*, 20(5):323–341, 2008.
21. Yujie Liu, Kunlong Zhang, and Michael Spear. Dynamic-sized Nonblocking Hash Tables. In *PODC*, pages 242–251, 2014.
22. Maged M. Michael. High Performance Dynamic Lock-Free Hash Tables and List-Based Sets. In *SPAA*, pages 73–82, 2002.
23. Jacob Nelson, Ahmed Hassan, and Roberto Palmieri. Bundled references: an abstraction for highly-concurrent linearizable range queries. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 448–450, 2021.
24. Erez Petrank and Shahar Timnat. Lock-free data-structure iterators. In *International Symposium on Distributed Computing*, 2013.

25. Julian Shun and Guy E. Blelloch. Ligra: a lightweight graph processing framework for shared memory. *SIGPLAN Not.*, 48(8):135–146, feb 2013. doi:10.1145/2517327.2442530.
26. Shahar Timnat, Anastasia Braginsky, Alex Kogan, and Erez Petrank. Wait-Free Linked-Lists. In *OPODIS*, pages 330–344, 2012.
27. Yuanhao Wei, Naama Ben-David, Guy E Blelloch, Panagiota Fatourou, Eric Ruppert, and Yihan Sun. Constant-time snapshots with applications to concurrent data structures. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 31–46, 2021.
28. Gerhard Weikum and Gottfried Vossen. *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery*. Morgan Kaufmann, 2002.
29. Kunlong Zhang, Yujiao Zhao, Yajun Yang, Yujie Liu, and Michael F. Spear. Practical Non-blocking Unordered Lists. In *DISC*, pages 239–253, 2013.