A Simple and Practical Concurrent Non-blocking Unbounded Graph with Linearizable Reachability Queries*

Bapi Chatterjee* Sathya Peri† Muktikanta Sa† Nandini Singhal‡†
*Institute of Science and Technology, Austria, bhaskerchatterjee@gmail.com
†Department of Computer Science & Engineering,
Indian Institute of Technology Hyderabad, India, {sathya_p, cs15resch11012}@iith.ac.in
‡Microsoft (R&D) Pvt. Ltd., Bangalore, India, nandini12396@gmail.com

ABSTRACT

Graph algorithms applied in many applications, including social networks, communication networks, VLSI design, graphics, and several others, require dynamic modifications – addition and removal of vertices and/or edges – in the graph. This paper presents a novel concurrent non-blocking algorithm to implement a dynamic unbounded directed graph in a shared-memory machine. The addition and removal operations of vertices and edges are lock-free. For a finite sized graph, the lookup operations are wait-free. Most significant component of the presented algorithm is the reachability query in a concurrent graph. The reachability queries in our algorithm are obstruction-free and thus impose minimal additional synchronization cost over other operations. We prove that each of the data structure operations are linearizable. We extensively evaluate a sample C/C++ implementation of the algorithm through a number of micro-benchmarks. The experimental results show that the proposed algorithm scales well with the number of threads and on an average provides 5 to 7x performance improvement over a concurrent graph implementation using coarse-grained locking.

CCS CONCEPTS

• **Computing methodologies** → **Shared memory algorithms**; **Concurrent algorithms**;

KEYWORDS

concurrent data structure, linearizability, linearization points, lock-free, wait-free, directed graph, reachable path

1 INTRODUCTION

1.1 Background

A *graph* is a highly useful data structure that models the pairwise relationships among real-world objects. Formally, it is represented as an ordered pair $G = (V, E)$, where V is the set of vertices and E is the set of edges. They underlay a number of important applications such as various kinds of networks (social, semantic, genomics, etc.), VLSI design, graphics, and several others. Generally, these applications require modifications such as insertion and deletion of vertices and/or edges in the graph to make them *dynamic*. Furthermore, they also require the data structure to grow at the run-time. The rise of multi-core systems has necessitated efficient and correct design of concurrent data structures, which can take advantage of the multi-threaded implementations. Naturally, a large number of applications would significantly benefit from concurrent dynamic unbounded graphs implemented on ubiquitous multi-core computers.

Arguably, the most important application of such a graph is performing a *reachability query*: for a given pair of vertices $u, v \in V$, determine if a sequence of adjacent vertices, i.e. a path, exists in V that starts at u and ends at v. In many use-cases, a reachability query requires returning the path if it exists. Performing a reachability query effectively entails exploring all the possible paths starting at u. In a dynamic and concurrent setting, where both V and E can be modified by a concurrent thread, it is extremely challenging to return a *valid* path, or for that matter being assured that there does not exist a path at the return of a reachability query. Besides, deletion of a vertex $v \in V$ involves deleting it along with all its incoming and outgoing edges in E. Obviously, a look-up or an insertion operation with a concurrent deletion poses complex design issues with regards to their correctness.

A well-accepted correctness-criterion for concurrent data structures is *linearizability* [8]. Broadly, a provably linearizable operation is perceived by a user as if it takes effect instantaneously at a point between its invocation and response during any concurrent execution. A simple and popular approach to handle updates in a concurrent data structure, while ensuring linearizability, is by way of mutual exclusion using *locks*. In case of a reachability query, that would essentially amount to locking the entire graph at its invocation and releasing the lock only at the return. However, in an asynchronous shared-memory system, such an implementation is vulnerable to arbitrary delays due to locks acquired by a slow

thread in addition to several other pitfalls such as deadlock, priority inversion and convoying.

In contrast, the *non-blocking* implementations – *wait-free*, *lock-free*, and *obstruction-free* [6, 7] – do not use locks and therefore are free from the drawbacks mentioned above. A wait-free operation in a concurrent data structure always finishes in a finite number of steps taken by a non-faulty thread. However, such a strong progress guarantee is too costly to implement. Alternatively, a lock-free operation is guaranteed to finish in a finite number of steps taken by some non-faulty thread. A concurrent data structure with all its operations satisfying lock-freedom is generally scalable. However, ensuring that each of the operations in a complex data structure, such as a graph, finishes in a finite number of steps by some thread, is still extremely challenging. Though weaker than the lock-freedom, the obstruction-freedom is still good enough to avoid the pitfalls of locks: an obstruction-free operation always finishes in a finite number of steps taken by any non-faulty thread running in isolation. Importantly, for obstruction-free operations, the design complexity as well as the synchronization overhead are much lower in comparison to their lock-free counterparts.

1.2 Our contributions

In this paper, we present a novel non-blocking algorithm for an unbounded directed[a] graph data structure. The contributions of this work are summarized below:

(1) For a directed graph $G = (V, E)$, we describe an Abstract Data Type (ADT) comprising of *modifications* – *addition*, *removal* – and *lookup* operations on the sets V and E. The ADT also comprises an operation to perform the *reachability queries* in G. The graph is represented as an adjacency list which enables it to grow without bound (up to the availability of memory) and sink at the runtime. (Section 2)

(2) We describe an efficient concurrent non-blocking implementation of the ADT (Section 3). To our knowledge, this is the first work on a non-blocking unbounded graph. The spotlight of our work is an obstruction-free reachability query. (Section 3.5)

(3) We prove the correctness in terms of the linearizability of the ADT operations. We also prove the non-blocking progress guarantee: (a) the modifications and lookup on vertices and edges of the graph are lock-free, (b) the reachability queries are obstruction-free, and (c) if the graph size is finite, the vertex and edge lookup operations are wait-free.(Section 4)

(4) For an experimental validation, we implemented the algorithm in C/C++. For comparison, we also implemented a sequential and a coarse-grained lock-based concurrent graph. We tested the implementations using a number of micro-benchmarks simulating varying workloads. The experiments demonstrate that the lock-free algorithm is highly scalable with the number of threads. We observed up to $5 - 7x$ higher throughput utilizing the available threads in our multi-core workstation while comparing the non-blocking algorithm with its sequential and coarse-grained lock-based counterparts. (Section 5)

---

[a]An undirected graph can be directly extended from a directed graph.

1.3 Related work

The concurrent graph data structure is largely an unexplored topic. There has been a recent interesting and relevant work by Kallimanis and Kanellou [9]. They proposed a concurrent graph that supports wait-free edge modifications and traversals. Their algorithm works on an adjacency matrix representation. They do not allow addition or removal of vertices after initialization of the graph, which renders it unsuitable for many applications that require dynamic and unbounded graphs. Moreover, it is not clear how the wait-free traversal will return a set of adjacent vertices in their algorithm.

1.4 A brief overview of the design

We implement the adjacency list of a directed graph as a sorted linked-list of *vertex-nodes*, where each of the vertex-node roots a sorted linked-list of *edge-nodes*, see Figure 1. The edge-nodes maintain pointers to the corresponding vertex-nodes to enable efficient graph traversals. The individual edge-node-lists and the vertex-node-list are lock-free with regards to the modifications and lookup operations.

The lock-free operations in the graph intuitively appear as a composition of the lock-free operations in the sorted vertex-list and the edge-lists. However, it is well-known that the lock-freedom is not composable [4], the progress guarantee of our algorithm is proved independent of the lock-free property of the component linked-lists. Furthermore, we propose some elegant optimizations in the operations' synchronization that not only ensure provable linearizability but also bring simplicity in the design.

For reachability queries we perform breadth first search (BFS) traversals in the graph. We implement the BFS traversals fully non-recursively for efficiency in a concurrent setting. It is natural that a reachability query is much more costlier compared to a modification or a lookup operation. However, in a concurrent setting it needs to synchronize with other concurrent operations. To ensure that the overall performance does not suffer from large reachability queries, we do not employ other operations to help them. Instead, to achieve the linearizability, we repeatedly collect concise versions of the graph and validate them by matching the return of two consecutive collections. Our approach is essentially based on the *double collect* [7, Chap 4] aided with several interesting optimizations. This design choice results in obstruction-free progress guarantee for the reachability queries.

2 THE GRAPH DATA STRUCTURE

2.1 The ADT

An abstract directed graph is given as $G = (V, E)$, where V is the set of *vertices* and E is the set of directed *edges* (ordered pair of vertices). Each edge connects an ordered pair of vertices belonging to V. A $v \in V$ maintains an immutable unique key $k \in K$, where K is a totally ordered set. A vertex $v \in V$ with key k is denoted by $v(k)$. We use $e(k, l)$ to denote an edge $(v(k), v(l)) \in E$.

We define an ADT for operations on G as given below.

(1) The ADDVERTEX(k) operation adds a vertex $v(k)$ to V, if $v(k) \notin V$ and returns true. If $v(k) \in V$, it returns false.

(2) The REMOVEVERTEX(k) operation removes $v(k)$ from V, if $v(k) \in V$ and returns true. If $v(k) \notin V$, it returns false. A

successful RemoveVertex(k) ensures that all $(j, k), (k, l) \in E$ are removed as well.

(3) The ContainsVertex(k) returns true, if $v(k) \in V$; otherwise, it returns false.

(4) The AddEdge(k, l) operation adds an edge $e(k, l)$ to E, if (a) $e(k, l) \notin E$, (b) $v(k) \in V$, and (c) $v(l) \in V$. If either of the conditions (a), (b) or (c) not satisfied, no change is made in E. For clarity about the reason of failure in adding an edge, we use different *indicative strings* for the different return cases.

(5) The RemoveEdge(k, l) operation removes the edge $e(k, l)$ if $e(k, l) \in E$. If $e(k, l) \notin E$, it makes no change in E. Similar to AddEdge, a RemoveEdge returns strings indicating if $v(k) \notin V$ or $v(l) \notin V$ or $e(k, l) \notin E$.

(6) The ContainsEdge(k, l) operation returns an indicative string "EDGE PRESENT " if $e(k, l) \in E$; otherwise, it returns similar strings as a RemoveEdge.

(7) The GetPath(k, l) operation returns a sequence of vertices – called a *path* – $\{v_i\}_{i=1}^{p} \subseteq V$, if (a) $v(k) \in V$, (b) $v(l) \in V$ (c) $(v(k), v_1) \in E$, (c) $(v_p, v(l)) \in E$, and (d) $(v_i, v_{i+1}) \in E \; \forall 1 \le i \le p$; otherwise, it returns NULL.
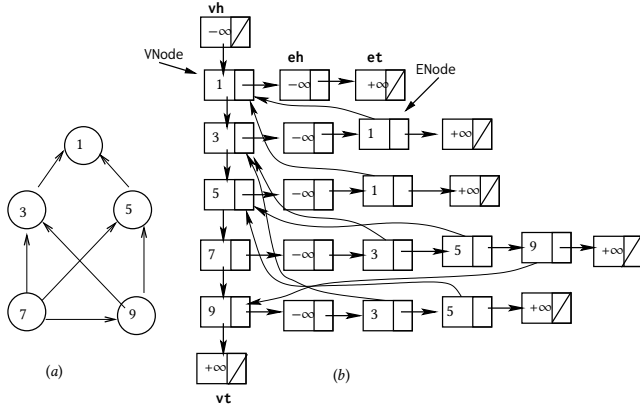
2.2 The data structure

Figure 1: (a) A directed Graph (b) The concurrent graph data structure for (a).

The above ADT is implemented by a data structure based on dynamically allocated *nodes*. Nodes are connected by word-sized *pointers*. The data structure essentially implements the *adjacency list* of G. See Figure 1. In our design an adjacency list is a composite linked-list of linked-lists. We call the composite linked-list the *vertex-list*. Each node in the vertex-list, called a VNode, corresponds to a $v(k) \in V$ and roots a linked-list called an *edge-list*. An edge-list rooted at a VNode corresponding to $v(k) \in V$ consists of nodes, called ENode, that correspond to the directed edges $e(k, l) \in E$.

The structures of the VNode and ENode are given in Figure 2. A VNode $v(k)$ consists of two pointers vnxt and enxt in addition to an immutable key k. vnxt points to the next VNode in the vertex-list, whereas, enxt points to the head of the edge-list rooted at $v(k)$. It also contains an array VisitedArray and an atomic counter ecnt. The array and the counter are used to facilitate BFS traversals, which we describe later. In the edge-list of $v(k)$, an ENode $e(k, l)$ contains two pointers ptv and enxt in addition to an immutable

key field l. We do not need to record the key of $v(k)$ in an ENode $e(k, l)$ because the ENodes are confined to the context of a single edge-list. ptv of $e(k, l)$ points to the VNode $v(l)$, whereas, its enxt points to the next ENode in the edge-list.

```
class VNode {                             class BFSNode {
    int k;              class ENode {          VNode n;
    VNode vnxt;             int l;             BFSNode nxt;
    ENode enxt;            VNode ptv;          BFSNode p;
    int VisitedArray[];    ENode enxt;     }
    int ecnt;          }
}
```

Figure 2: Structure of ENode, VNode and BFSNode.

To avoid dereferencing the null pointers, we initialize the vertex-list with sentinel nodes $v(-\infty)$ and $v(\infty)$ representing dummy head and tail, respectively. Similarly, each of the edge-lists are initialized with $e(-\infty)$ and $e(\infty)$ representing its dummy head and tail, respectively.

The data structure maintains simple *invariants*: (a) the vertex-list is sorted, and (b) each of the edge-lists are sorted.

The implementation of the ADT operations in the data structure as described here is called their *sequential specification*. Traversals in the vertex-list and edge-lists are performed following their sorted order starting from the dummy head and stopping at an appropriate node. AddVertex(k), RemoveVertex(k) and ContainsVertex(k) locate $v(k)$ traversing down the vertex-list starting from $v(-\infty)$ and stopping at a node $v(j)$ such that $j \le k$ and $\nexists v(j') \in V$ where $j < j' \le k$. The pointer modifications by AddVertex and RemoveVertex are exactly same as in a sequential linked-list data structure [5]. ContainsVertex returns true if $j = k$ else it returns false.

The AddEdge(k, l), RemoveEdge(k, l) and ContainsEdge(k, l) first locate both $v(k)$ and $v(l)$ in the vertex-list. Only if both the nodes are located in the vertex-list, the next step is taken. In the next step they traverse down the edge-list of $v(k)$. Addition, removal and lookup in an edge-list are similar to those in the vertex-list. Interestingly, when the VNode $v(l)$ is removed from the vertex-list, we may not necessarily have to remove $e(k, l)$ from the edge-list of $v(k)$, and still a ContainsEdge(k, l) will work correctly because it would not traverse down the edge-list of $v(k)$ after finding that $v(l)$ does not exist in the vertex-list. The return of these operations are indicative strings as mentioned in the ADT definition.

A GetPath(k, l) operation first locates $v(k)$ and $v(l)$ in the vertex-list. If either of them is not located, it right away returns NULL. On locating $v(k)$, it performs a BFS traversal in the data structure starting at $v(k)$ [3]. The pointers ptv of ENodes help a traversal to directly go to a VNode from an ENode. An integer (for a dirty bit as described in [3]) is used to keep track of the visited nodes during BFS traversals. The array VisitedArray in a VNode serves this purpose in concurrent multi-threaded implementations. A BFS traversal for GetPath(k, l) returns as soon as the node $v(l)$ is located.

The return of a BFS is a linked-list of BFSNodes, see Figure 2. A BFSNode $b(v)$, corresponding to a visited VNode v, contains v. It also contains two BFSNode pointers nxt and p. The pointer nxt points to the next node in the returned linked-list. p is a special pointer pointing to the BFSNode containing the VNode in the graph from which the BFS traverses to v. p facilitates the tracing of the path from $v(k)$ to $v(l)$ in the return of GetPath(k, l). Starting from

the last node in the returned list, which necessarily corresponds to $v(l)$ if there exists a path between $v(k)$ and $v(l)$, GETPATH(k, l) returns the sequence of the adjacent nodes. If the last BFSNode in the returned list does not contain $v(l)$, the return of GETPATH(k, l) is NULL.

3 THE NON-BLOCKING ALGORITHM

3.1 The shared-memory system

For a concurrent non-blocking implementation of the data structure, we consider a *shared-memory system* consisting of a finite set of *processors* accessed by a finite set of *threads* that run in a completely asynchronous manner. The threads communicate with each other by invoking operations on shared objects and getting corresponding responses. The pointers and other fields of the various nodes are implemented by the shared objects. The system supports atomic read, write, fetch-and-add (FAA) and compare-and-swap (CAS) instructions.

A FAA (*address*, *val*) atomically increments the value at the memory location *address* by *val*. A CAS(*address*, *old*, *new*) instruction checks if the current value at a memory location *address* is equivalent to the given value *old*, and only if true, changes the value of *address* to the new value *new* and returns true; otherwise the memory location remains unchanged and the instruction returns false. Such a system can be perfectly realized by a Non-Uniform Memory Access (NUMA) computer with one or more multi-processor CPUs.

3.2 The design basics

The basic structure of the presented graph data structure is based on a linked-list. Therefore, for the lock-free synchronization in the graph, we utilize the approach of an existing lock-free linked-list algorithm [5]. The core idea of the design is a remove procedure based on a protocol of first atomically injecting an *operation descriptor* on the outgoing pointers of the VNodes or ENodes, which are to be removed, and then atomically modifying the incoming pointers to disconnect the nodes from the vertex-list or edge-lists. If multiple concurrent operations try to modify a pointer simultaneously, they synchronize by helping the pending removal operation that would have successfully injected its descriptor.

More specifically, to remove a VNode (respectively ENode) n from the vertex-list (respectively an edge-list), we use a CAS to inject an operation descriptor at the pointer vnxt (respectively enxt). We call these descriptors a *mark* and a pointer with a descriptor as *marked*. We call a VNode (respectively ENode) *marked* if it vnxt (respectively enxt) pointer is marked. A pointer once marked is never modified again.

A concurrent operation, if obstructed at a marked pointer, helps by performing the remaining step of a removal: modifying the incoming pointer from the previous node to point the next node in the list and thereby removing the node. An addition operation uses a single CAS to update the target pointer only if it is not marked, called *clean*, otherwise it helps the pending removal operation. During lookup or a reachability query, a traversal on the vertex-list or the edge-lists does not perform any help. Traversals for modification operations help pending removal operations. After helping a pending removal operation, a concurrent addition or removal operation restart suitably.

To realize the atomic step to inject an operation descriptor, we replace a pointer using a CAS with a single-word-sized packet of itself and an operation descriptor. In C/C++, to pack the operation descriptor with a pointer in a single memory-word, we apply the so-called *bit-stealing*. In a x86/64 machine, where memory allocation is aligned on a 64-bit boundary, three least significant bits in a pointer are unused. The mark descriptor uses the last significant bit: if the bit is set the pointer is marked, otherwise clean.

For ease of exposition, we assume that a memory allocator always allocates a variable at a new address and thus an ABA problem does not occur. ABA is an acronym to indicate a typical problem in a CAS-based lock-free algorithm: a value at a shared variable can change from A to B and then back to A, which can corrupt the semantics of the algorithm. We assume the availability of a lock-free memory reclamation scheme.

Pseudo-code convention: The algorithm is presented in the pseudo-codes in the Figures 3, 4 and 6. If x is a pointer pointing to a class instance, we use $x.y$ to indicate the field y of the instance of the class. $< r_1, r_2, \ldots, r_n >$ indicates a return of multiple variables together. For a pointer x, MRKDRF(x) and UNMRKDRF(x) denote x with its last significant bit set to 1 and 0, respectively. ISMRKD(x) returns true if the last significant bit of x is set to 1, otherwise, it returns false. A call of CVNODE(k) instantiates a new VNode with key k, whereas, CENODE(l) instantiates a new ENode with key l. For a newly instantiated VNode or ENode, the pointer fields are NULL, integer array has 0 in each slot and an integer counter is 0.

3.3 The lock-free vertex operations

The operations ADDVERTEX, REMOVEVERTEX and CONTAINSVERTEX are shown in the Figure 3. Fundamentally, these operations in our algorithm are similar to the ones in [5]. However, unlike [5], CONTAINSVERTEX does not indulge in helping during traversal. This essentially makes the CONTAINSVERTEX operations wait-free in case the set of keys is finite. For the sake of efficiency. We believe that CONTAINSVERTEX operation will be lot more frequent than update operations. Hence, we are not making the CONTAINSVERTEX operation help the update operations. Inspite of this, the algorithms are still lock-free. The return of the operations are as described in their respective sequential specifications presented in the Section 2.2.

An ADDVERTEX (k) operation, in the line 1 to 12, first calls LOCV procedure to locate the appropriate VNode, ahead of which it needs to add the new VNode. On locating a clean vnxt pointer of a VNode with the key just less than k, it attempts a CAS to add the new VNode. On a CAS failure, the process is reattempted. A REMOVEVERTEX (k), line 13 to 29, similarly traverses down the vertex-list by calling LOCV. On locating the VNode $v(k)$ to remove, it (a) marks the vnxt of $v(k)$ using a CAS, and (b) atomically updates the vnxt of the previous node in the vertex-list to point to the next VNode after $v(k)$ using a CAS. On any CAS failure the process is reattempted.

During traversal in a LOCV, line 101 to 117, we help a pending REMOVEVERTEX by essentially completing the step (b) as described above.

A successful CAS at line 21 is called the *logical removal* of $v(k)$. After this step any call of ISMRKD $(v(k).$vnxt$)$ would return false,

```
 1: Operation ADDVERTEX (k)
 2:   while (true) do
 3:     ⟨pv, cv⟩ ← LOCV (vh, k);
 4:     if (cv.k = k) then return false;
 5:     else
 6:       nv ← CVNODE (k); nv.vnxt ← cv;
 7:       if (CAS(pv.vnxt, cv, nv)) then
 8:         return true;
 9:       end if
10:     end if
11:   end while
12: end Operation

13: Operation REMOVEVERTEX(k)
14:   while (true) do
15:     ⟨pv, cv⟩ ← LOCV (vh, k);
16:     if (cv.k ≠ k) then
17:       return false;
18:     end if
19:     cn ← cv.vnxt;
20:     if (¬ ISMRKD (cn)) then
21:       if (CAS(cv.vnxt, cn, MRKDRF (cn)))
       then
22:         if (CAS(pv.vnxt, cv, cn)) then
23:           break;
24:         end if
25:       end if
26:     end if
27:   end while
28:   return true;
29: end Operation

30: Operation CONTAINSVERTEX(k)
31:   cv ← vh.vnxt;
32:   while (cv.k < k) do
33:     cv ← UNMRKDRF (cv.vnxt);
34:   end while
35:   if (cv.k = k ∧ ¬ ISMRKD (cv)) then
36:     return true;
37:   else  return false;
38:   end if
39: end Operation
```

```
40: Operation ADDEDGE(k, l)
41:   ⟨ u, v, st ⟩ ← CONVPLUS(k, l);
42:   if (st = false) then
43:     return "VERTEX NOT PRESENT" ;
44:   end if
45:   while (true) do
46:     if (ISMRKD (u) ∨ ISMRKD (v)) then
47:       return "VERTEX NOT PRESENT";
48:     end if
49:     ⟨pe, ce⟩ ← LOCE (u.enxt, l);
50:     if (ce.l = l) then
51:       return "EDGE PRESENT";
52:     end if
53:     ne ← CENODE (l);
54:     ne.enxt ← ce;
55:     ne.ptv ← v;
56:     if (CAS(pe.enxt, ce, ne)) then
57:       u.ecnt.FetchAndAdd (1);
58:       return "EDGE ADDED";
59:     end if
60:   end while
61: end Operation

62: Operation CONTAINSEDGE(k, l)
63:   ⟨ u, v, st ⟩ ← CONCPLUS(k, l);
64:   if (st = false) then
65:     return "VERTEX NOT PRESENT";
66:   end if
67:   ce ← u.enxt;
68:   while (ce.l < l) do
69:     ce ← UNMRKDRF (ce.enxt);
70:   end while
71:   if (ce.l = l ∧ ¬ ISMRKD (u) ∧ ¬ ISMRKD
   (v) ∧ ¬ ISMRKD (ce)); then
72:     return "EDGE FOUND" ;
73:   else
74:     return "VERTEX  OR  EDGE  NOT
   PRESENT";
75:   end if
76: end Operation
```

```
 77: Operation REMOVEEDGE(k, l)
 78:   ⟨ u, v, st ⟩ ← CONVPLUS(k, l);
 79:   if (st = false) then
 80:     return "VERTEX NOT PRESENT";
 81:   end if
 82:   while (true) do
 83:     if (ISMRKD (u) ∨ ISMRKD (v)) then
 84:       return "VERTEX NOT PRESENT";
 85:     end if
 86:     ⟨pe, ce⟩ ← LOCE (u.enxt, l);
 87:     if (ce.l ≠ l) then
 88:       return "EDGE NOT PRESENT";
 89:     end if
 90:     cnt ← ce.enxt;
 91:     if (¬ ISMRKD (cnt)) then
 92:       if (CAS(ce.enxt, cnt, MRKDRF (cnt))) then
 93:         u.ecnt.FetchAndAdd(1);
 94:         if (CAS(pe.enxt, ce, cnt)) then  break;
 95:         end if
 96:       end if
 97:     end if
 98:   end while
 99:   return "EDGE REMOVED";
100: end Operation

101: procedure LOCV(v, k)
102:   while (true) do
103:     pv ← v; cv ← pv.vnxt;
104:     while (true) do
105:       cn ← cv.vnxt;
106:       while (ISMRKD (cn) ∧(cv.k < k)) do
107:         if (¬CAS(pv.vnxt, cv, cv.vnxt)) then
108:           goto 102;
109:         end if
110:         cv ← cn; cn ← UNMRKDRF(cv.vnxt);
111:       end while
112:       if (cv.k ≥ k) then return ⟨pv, cv⟩;
113:       end if
114:       pv ← cv; cv ← cn;
115:     end while
116:   end while
117: end procedure
```

Figure 3: Pseudo-codes of ADDVERTEX, REMOVEVERTEX, CONTAINSVERTEX, ADDEDGE, REMOVEEDGE, CONTAINSEDGE and LOCV.

which is used by a CONTAINSVERTEX, line 30 to line 39, to return false in case a marked $v(k)$ is located.

The removal of a vertex from a graph also requires removing all the incoming and outgoing edges of it. The outgoing edges are removed along with the VNode $v(k)$, as it is logically, and eventually *physically* detached from the vertex-list by a successful CAS at line 22. However, the incoming edges are logically removed as any ptv from an ENode of any edge-list would call ISMRKD to check the removed VNode. As an optimization, we leave the ENodes in all the edge-lists with their ptv pointing to $v(k)$ as they were. Eventually those ENodes are removed as part of the physical removal of the roots of their respective edge-lists. Note that, the physical removal can be performed by any helping operation.

3.4 The lock-free edge operations

An ADDEDGE(k, l) operation, line 40 to 61, starts by verifying the presence of vertices $v(k)$ and $v(l)$ in the vertex-list of the graph by invoking the CONVPLUS at the line 41. The procedure CONVPLUS, line 147 to line 168, essentially locates the VNode with the smaller key first starting from $v(-\infty)$ and if located, starts from that VNode to locate the VNode with the bigger key. If any of the nodes not located ADDEDGE(k, l) returns the string VERTEX NOT PRESENT. Once both the nodes located, it adds a new ENode $e(k, l)$ in the edge-list of $v(k)$ along the same lines of addition of a VNode in the vertex-list. To traverse down an edge-list, the procedure LOCE is called, see line 118 to 146 in the Figure 4.

A LOCE traverses down an edge-list and physically removes two kind of logically removed ENodes: (a) the ones corresponding to a logically removed VNode, see the lines 124 and 127, and (b) logically removed ENodes, see the line 134. It returns the address of

two consecutive ENodes between which the new ENode could be added. Before every attempt of executing a CAS to add an ENode, an ADDEDGE(k, l) checks if the VNodes $v(k)$ and $v(l)$ are logically removed. This check ensures avoiding an interesting wrong execution as illustrated in the Figure 5. If the ENode $e(k, l)$ is found in the edge-list of $v(k)$, the string EDGE PRESENT is returned. A successful CAS to add the ENode returns EDGE ADDED.

A REMOVEEDGE(k, l) operation, line 77 to line 100, works along the similar lines as ADDEDGE(k, l) during traversal to locate the VNodes $v(k)$ and $v(l)$ and the ENode $e(k, l)$. If all of them found in the data structure, it uses CAS to first logically remove the ENode, line 92, and thereafter physically remove the ENode, line 94. In case either of the vertices $v(k)$ or $v(l)$ not present in the data structure, it returns VERTEX NOT PRESENT. If the ENode $e(k, l)$ not found, it returns EDGE NOT PRESENT. On a successful CAS to logically remove an ENode, it returns EDGE REMOVED.

The operations ADDEDGE(k, l) and REMOVEEDGE(k, l) also increment an atomic counter at the node $v(k)$ using a FAA, see the lines 57 and 93. This atomic counter facilitates in comparing the output of consecutive BFS traversals in the GETPATH operation. We discuss it in the next section. A LOCE, as it can be called from an ADDEDGE or a REMOVEEDGE, also increments the atomic counter before physically removing an ENode, see line 133. It ensures that if a thread after logically removing an ENode got delayed then a helper thread increments the counter on its behalf.

A CONTAINSEDGE(k, l) operation, locates the VNodes $v(k)$, $v(l)$ and the ENodes $e(k, l)$ similar to other edge operations. Before returning an indicative string as appropriate, it ensures that non of the nodes $v(k)$, $v(l)$ and $e(k, l)$ are marked, see the lines 62 to 76. For the sake of efficiency. Like CONTAINSVERTEX, the CONTAINSEDGE also does not indulge in helping the update operations during traversal.

3.5 The obstruction-free GETPATH operation

The design of the GETPATH (see Figure 6) operation draws from the snapshot algorithm proposed by Afek et al. [1]. A GETPATH(k, l), line 200 to 215, first checks the presence of $v(k)$ and $v(l)$ by invoking the CONCPLUS procedure at the line 202. After successfully checking the presence of both the vertices it goes to perform repeated BFS traversals by invoking the procedure SCAN at the line 210. If the VNodes $v(k)$, $v(l)$ are not located, it right away returns NULL.

The SCAN procedure, line 216 to 230, first initializes two lists of BFSNodes. We call such a list a BFS-tree. The two BFS-trees are used to hold collection of VNodes in two consecutive BFS traversals. A reader not familiar with the BFS traversals in graphs may refer to any standard book on algorithms such as [3]. The procedure TREECOLLECT, line 231 to 261, takes in a BFS-tree and fills it with nodes collected in a BFS traversal. A BFS traversal terminates as soon as the VNode $v(l)$ is located. However, in case $v(l)$ could not be reached from $v(k)$, the traversal terminates after exhausting all the outgoing edges from $v(k)$ represented by the ENodes in its edge-list.

During a BFS traversal, we put markers on the VNodes to keep track of the visited ones, see [3]. In a sequential implementation, a single boolean variable is good enough. However, in our case, when multiple threads perform BFS traversals not only concurrently but also repeatedly, a single boolean variable or for that matter an boolean array would not suffice. To keep track of visited VNodes, we use the array VisitedArray in them. The size of VisitedArray

is equal to the number of threads in the shared-memory system. Thus, a slot of VisitedArray, used as a counter for the number of visits, provides local marker for repeated traversals by a thread.

The return of the TREECOLLECT procedure is a boolean indicating if $v(l)$ was located. If the return of two consecutive TREECOLLECT do not match we discard the old BFS-tree and start collecting a new one, see the line 228. However, if the returns match and both are true, it indicates that both the times a path from $v(k)$ to $v(l)$ could be discovered. Hence, we compare the collected paths, which are subset of the two BFS-trees. Please notice that we can not return either of them unless the two paths match because we are not sure if either existed at any instant during the lifetime of GETPATH.

The procedure COMPAREPATH, line 276 to 289, compares two BFS-trees with respect to the paths between $v(k)$ and $v(l)$: it starts from the last BFSNodes in the two BFS-trees and follows the BFSNode-pointers p that takes to the previous node in a possible path; at any BFSNode if there is a mismatch between the contained VNodes, it terminates.

If the returns of two consecutive TREECOLLECT are false, it indicates that both the times a path was not traced between $v(k)$ and $v(l)$. However, to be sure that during the entire lifetime of the GETPATH(k, l), at every point in time no path ever existed, we need to compare the two returned BFS-trees. The comparison of the two BFS-trees in entirety is done in the procedure COMPARETREE, see the line 262 to 275.

If the comparison of two consecutive BFS-trees do not match in procedures COMPARETREE or COMPAREPATH, we discard the first BFS-tree and restart the tree-collection.

While comparing two BFS-trees or the paths therein in the procedures COMPARETREE or COMPAREPATH, the BFS-trees require to be compared with respect to not just the BFSNodes, but also the counters ecnt of the VNodes contained in them. We explain this requirement below.

Consider an adversary against a GETPATH operation. Consider two consecutive BFS traversals. Suppose that during the first traversal after the TREECOLLECT discovered that no path to $v(l)$ existed via a VNode v_i and therefore moved to another VNode v_{i+1} and continued the traversal until its exploration exhausted. As we know that once v_i is visited, it will not be revisited. Now suppose that when TREECOLLECT was visiting nodes after v_{i+1}, the adversary added an edge $(v_i, v(l))$ that made a path exist between $v(k)$ and $v(l)$. However, before we could start the second BFS traversal, the edge $(v_i, v(l))$ was removed by the adversary bringing the graph exactly at the same state at which the first traversal had started. Now, suppose that even during the second traversal the same game is played by the adversary. In such a scenario, if we just matched the two consecutive BFS-trees with respect the the collected BFSNodes, the operation GETPATH(k, l) would return NULL indicating that a path did not exist during its lifetime, which would be clearly incorrect.

During the edge modification operations, before an ENode $(v_i, v(l))$ is physically removed, the atomic counter ecnt at v_i is necessarily incremented by either the operation that logically removed $(v_i, v(l))$ or a helping operation at the lines 93 or 57 or 133. This ensures that we get to check the adversaries as described above. Although, it may make a GETPATH continue until all the modification operations in the graph stop, we still have an ensured correct return of a reachability query.

```
118: procedure locE(v, k)
119:   while (true) do
120:     pe ← v; ce ← pe.enxt;
121:     while (true) do
122:       cnt ← ce.enxt; VNode vn ← ce.ptv;
123:       while (isMrkd (vn) ∧ ¬ isMrkd (cnt)) do
124:         if (¬CAS(ce.enxt, cnt, MrkdRf (cnt))) then
125:           goto Line 119;
126:         end if
127:         if (¬CAS(pe.enxt, ce, cnt)) then goto Line 119;
128:         end if
129:         ce ← cnt; vn ← ce.ptv;
130:         cnt ← UnMrkdRf(ce.enxt);
131:       end while
132:       while (isMrkd (cnt)) do
133:         v.ecnt.FetchAndAdd (1);
134:         if (¬ CAS(pe.enxt, ce, cnt)); then goto 119;
135:         end if
136:         ce ← cnt; vn ← ce.ptv;
137:         cnt ← UnMrkdRf(ce.enxt);
138:       end while
139:       if (isMrkd (vn)) then  goto Line 123;
140:       end if
141:       if (ce.l ≥ k) then return ⟨pe, ce⟩
142:       end if
143:       pe ← ce; ce ← cnt;
144:     end while
145:   end while
146: end procedure
```

```
147: procedure ConVPlus (k, l)
148:   if (k < l) then
149:     ⟨pv1, cv1⟩ ← locV(vh, k);
150:     if (cv1.k ≠ k) then
151:       return ⟨NULL, NULL, false⟩;
152:     end if
153:     ⟨pv2, cv2⟩ ← locV(cv1, l);
154:     if (cv2.k ≠ l) then
155:       return ⟨NULL, NULL, false⟩;
156:     end if
157:   else
158:     ⟨pv2, cv2⟩ ← locV(vh, l);
159:     if (cv2.k ≠ l) then
160:       return ⟨NULL, NULL, false⟩;
161:     end if
162:     ⟨pv1, cv1⟩ ← locV(cv2, k);
163:     if (cv1.k ≠ k) then
164:       return ⟨NULL, NULL, false⟩ ;
165:     end if
166:   end if
167:   returns ⟨cv1, cv2, true⟩;
168: end procedure
```

```
169: procedure locC(v, k)
170:   pv ← v; cv ← pv.vnxt;
171:   while (true) do
172:     if (cv.k ≥ k) then
173:       return ⟨pv, cv⟩;
174:     end if
175:     pv ← cv; cv ← UnMrkdRf(cv.vnxt);
176:   end while
177: end procedure
```

```
178: procedure ConCPlus (k, l)
179:   if (k < l) then
180:     ⟨pv1, cv1⟩ ← locC(vh, k);
181:     if (cv1.k ≠ k) then
182:       return
           ⟨NULL, NULL, false⟩;
183:     end if
184:     ⟨pv2, cv2⟩ ← locC(cv1, l);
185:     if (cv2.k ≠ 2) then
186:       return
           ⟨NULL, NULL, false⟩;
187:     end if
188:   else
189:     ⟨pv2, cv2⟩ ← locC(vh, l);
190:     if (cv2.k ≠ l) then
191:       return
           ⟨NULL, NULL, false⟩;
192:     end if
193:     ⟨pv1, cv1⟩ ←  locC(cv2,
     k);
194:     if (cv1.k ≠ k) then
195:       return
           ⟨NULL, NULL, false⟩ ;
196:     end if
197:   end if
198:   returns ⟨cv1, cv2, true⟩;
199: end procedure
```
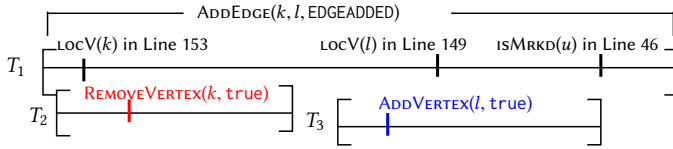
Figure 4: Pseudo-codes of locE, ConCPlus, locC and ConCPlus.

Figure 5: The reason we need to check (Line 46) $v(k)$ or $v(l)$ is marked or not in AddEdge method. Thread T_1 trying to perform AddEdge(k, l), first invokes locC(k). After T_1 has verified $v(k)$ to be present in vertex-list, thread T_2 deletes $v(k)$ and thread T_3 adds $v(l)$. In the absence of Line 46 check, T_1 confirms $v(l)$ to be present and adds an edge (k, l), thereby returning EDGE ADDED. However, this is an **illegal execution**. In no possible sequential history equivalent to the given concurrent execution will both the vertices $v(k)$ and $v(l)$ exist together. By having an check in this scenario, AddEdge(k, l) will return VERTEX NOT PRESENT on checking that vertex $v(k)$ has been deleted.

4 CORRECTNESS AND PROGRESS

4.1 Linearizability

The ADT operations implemented by the data structure are represented by their invocation and return steps. To prove the correctness of an arbitrary concurrent execution of a set of ADT operations, we show that they satisfy the consistency framework *linearizability* [8]. To do that, firstly we show that it is possible to assign an atomic step as a *linearization point* (LP) inside the execution interval of each of the operations. Thereafter, we also show that the data structure invariants are maintained across the LPs. Thereby, it proves that an arbitrary concurrent execution is equivalent to a valid sequential execution obtained by ordering the operations by their LPs.

THEOREM 4.1. *The ADT operations implemented by the presented algorithm are linearizable.*

PROOF. For ease of presentation, we discuss the LPs in case by case manner depending on the return of the operations.

(1) AddVertex(k): Two cases based on the return values:
   (a) true: Successful compare-and-swap execution at the line 7.
   (b) false: The vertex is already present. The LP is the atomic read of the vnxt pointer pointing to $v(k)$.
(2) RemoveVertex(k): Two cases based on the return values:
   (a) true: Successful CAS execution at the line 21 (logical removal).
   (b) false: If a concurrent RemoveVertex operation *op* removed $v(k)$ then just after the LP of *op*. $v(k)$ did not exist in the vertex-list then at the invocation of RemoveVertex(k).
(3) ContainsVertex(k): Two cases based on the return value:
   (a) true: The atomic read of the vnxt pointing to $v(k)$.
   (b) false: same as the case 2b, where RemoveVertex returns false.
(4) AddEdge(k, l): Three cases depending on the return values:
   (a) EDGE ADDED: Two sub-cases depending on if there is a concurrent RemoveVertex(k) or a RemoveVertex(l):
      (i) No concurrent RemoveVertex(k) or RemoveVertex(l): the successful CAS execution at the line 56.

```
200: Operation GetPath (k, l)
201:    tid ← this_thread.get_id();
202:    ⟨ u, v, st ⟩ ← ConCPlus(k, l);
203:    if (st = false) then
204:      return NULL;
205:    end if
206:    if (isMrkd (u) ⋁ isMrkd (v)) then
207:      return NULL;
208:    end if
209:    list < BFSNode > bTree;
210:    ⟨ st, bTree ⟩ ← Scan (u, v, tid);
211:    if (st = true) then
212:      return "PATH u TO v FROM THE bTree";
213:    else   return NULL;
214:    end if
215: end Operation
216: procedure Scan (u, v, tid)
217:    list < BFSNode > ot, nt ;
218:    bool f1 ← TreeCollect (u, v, ot, tid);
219:    while (true) do
220:      bool f2 ← TreeCollect (u, v, nt, tid);
221:      if (f1 = true ⋀ f2 = true ⋀ ComparePath
         (ot, nt)) then
222:        return ⟨f2, nt⟩;
223:      else
224:        if (f1 = false ⋀ f2 = false ⋀ Compare-
         Tree (ot, nt)) then
225:          return ⟨f2, NULL ⟩;
226:        end if
227:      end if
228:      f1 ← f2; ot ← nt;
229:    end while
230: end procedure
```

```
231: procedure TreeCollect (u, v, bTree, tid)
232:    queue <BFSNode > que;
233:    cnt ← cnt + 1;
234:    u.visitedArray[tid] ← cnt ;
235:    BFSNode bNode(u, NULL, NULL);
236:    bTree.Insert(bNode);
237:    que.enque(bNode);
238:    while (¬que.empty()) do
239:      BFSNode cvn ← que.deque();
240:      eh ← cvn.n.enext;
241:      for (ENode itn ← eh.enxt to itn.enxt ≠
         NULL) do
242:        if (¬isMrkd (itn)) then
243:          VNode adjn ← itn.ptv;
244:          if (¬isMrkd (adjn)) then
245:            if (adjn = v) then
246:              BFSNode bNode(adjn, cvn, NULL);
247:              bTree.Insert(bNode);
248:              return true;
249:            end if
250:            if (¬ CheckVisited (tid, adjn, cnt))
         then
251:              adjn.VisitedArray [tid] ← cnt ;
252:              BFSNode bNode(adjn, cvn, NULL);
253:              bTree.Insert(bNode);
254:              que.enque(bNode);
255:            end if
256:          end if
257:        end if
258:      end for
259:    end while
260:    return false;
261: end procedure
```

```
262: procedure CompareTree (ot, nt)
263:    if (ot = NULL ⋁ nt = NULL) then
264:      return false ;
265:    end if
266:    BFSNode oit ← ot.Head, nit ← nt.Head;
267:    do
268:      if (oit.n ≠ nit.n ⋁ oit.ecnt ≠ nit.ecnt ⋁
         oldit.p ≠ newit.p) then   return false;
269:      end if
270:      oit ← oit.nxt; nit ← nit.nxt;
271:    while (oit ≠ ot.Tail ⋀ nit ≠ nt.Tail );
272:    if (oit.n ≠ nit.n ⋁ oit.ecnt ≠ nit.ecnt ⋁ oit.p
         ≠ nit.p) then   return false;
273:    else   return true ;
274:    end if
275: end procedure
276: procedure ComparePath (ot, nt)
277:    if (ot = NULL ⋁ nt = NULL) then
278:      return false ;
279:    end if
280:    BFSNode oit ← ot.Tail, nit ← nt.Tail;
281:    do
282:      if (oit.n ≠ nit.n ⋁ oit.ecnt ≠ nit.ecnt ⋁
         oldit.p ≠ newit.p) then   return false;
283:      end if
284:      oit ← oit.p; nit ← nit.p;
285:    while (oit ≠ ot.Head ⋀ nit ≠ nt.Head );
286:    if (oit.n ≠ nit.n ⋁ oit.ecnt ≠ nit.ecnt ⋁ oit.p
         ≠ nit.p) then   return false;
287:    else   return true;
288:    end if
289: end procedure
```

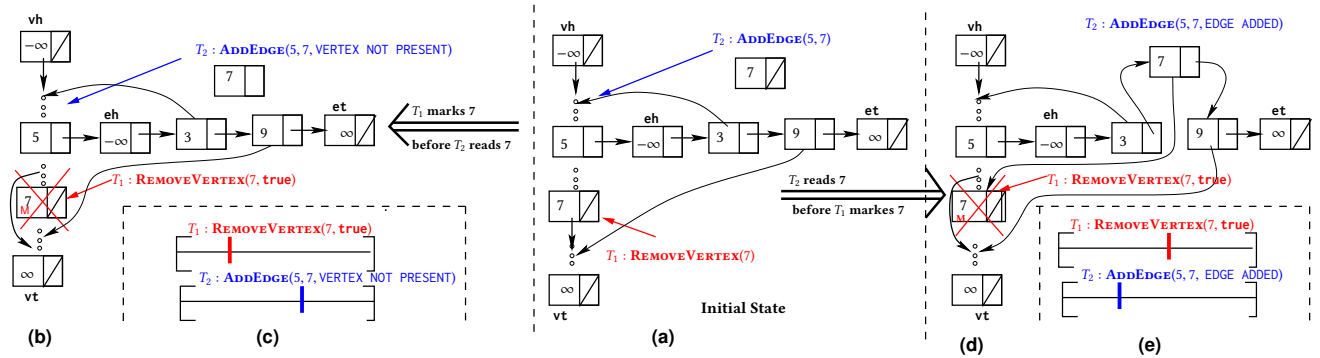Figure 6: Pseudo-codes of GetPath, Scan, TreeCollect, CompareTree and ComparePath.

Figure 7: Execution with two concurrent operations - AddEdge and RemoveVertex

(ii) With concurrent RemoveVertex(k) or RemoveVertex(l): just before the first remove's LP. A sample case for determining the LP when a RemoveVertex is concurrent to an AddEdge is shown in the Figure 7.

(b) EDGE PRESENT: Similar to the case when EDGE ADDED is returned.

(i) No concurrent RemoveVertex(k) or RemoveVertex(l) or RemoveEdge(k, l): the atomic read of the enxt pointer pointing to e(k, l).

(ii) With concurrent RemoveVertex(k) or RemoveVertex(l) or RemoveEdge(k, l): just before the first remove's (either vertex or edge) LP.

(c) VERTEX NOT PRESENT: Two sub-cases

(i) If both v(k) and v(l) were in the vertex-list at the invocation of AddEdge(k, l) and a concurrent RemoveVertex removed v(k) or v(l) or both then just after the LP of the earlier RemoveVertex.

(ii) If v(k) and/or v(l) were not present at the invocation of AddEdge(k, l) then the invocation point itself.

(5) REMOVEEDGE(k, l): Similar to ADDEDGE, we have three cases depending on the return values:
　(a) EDGE REMOVED: This is similar to the case 4a of ADDEDGE. We have two sub-cases depending on if there are any concurrent REMOVEVERTEX operations:
　　(i) No concurrent REMOVEVERTEX(k) or REMOVEVERTEX(l): the successful CAS execution at the Line 92.
　　(ii) With concurrent REMOVEVERTEX(k) or REMOVEVERTEX(l): just before the first remove's LP.
　(b) EDGE NOT PRESENT: If a concurrent REMOVEEDGE operation removed $e(k, l)$ then just after its LP, otherwise at the invocation of REMOVEEDGE(k, l) itself.
　(c) VERTEX NOT PRESENT: Absolutely same as the case ADDEDGE returning "VERTEX NOT PRESENT".
(6) CONTAINSEDGE(k, l): Similar to REMOVEEDGE, we have three cases depending on the return values. All the steps are very similar REMOVEEDGE.
　(a) EDGE PRESENT: We have two sub-cases depending on if there are any concurrent REMOVEVERTEX operations:
　　(i) No concurrent REMOVEVERTEX(k) or REMOVEVERTEX(l): the atomic read of the enxt pointer pointing to $e(k, l)$..
　　(ii) With concurrent REMOVEVERTEX(k) or REMOVEVERTEX(l): just before the LP of the earlier REMOVEVERTEX.
　(b) EDGE NOT PRESENT: Absolutely same as the case of REMOVEEDGE returning "EDGE NOT PRESENT".
　(c) VERTEX NOT PRESENT: Absolutely same as the case of ADDEDGE returning "VERTEX NOT PRESENT".
(7) GETPATH(k, l) : Here, there are two cases:
　(a) GETPATH invoke the SCAN operation: Assuming that SCAN invokes m (greater than equal to 2) TREECOLLECT procedures. Then it is the last atomic read step of the $(m − 1)^{st}$ TREECOLLECT call.
　(b) GETPATH does not invoke the SCAN operation: If a concurrent REMOVEVERTEX operation op removed $v(k)$ or $v(l)$. Then just after the LP of op. If $v(k)$ or $v(l)$ did not exist in the vertex-list before the invocation then at the invocation of GETPATH(k, l).

In the above discussion it is easy to observe that each of the LPs belong the interval between the invocation and return steps of the corresponding operations.

We can also observe in the algorithm that in any call of an ADDVERTEX(k), the traversal terminates at the VNode where the key is just less than or equal to k. Similar argument holds true for a call of ADDEDGE(k, l). Before every reattempt of ADDVERTEX(k) and ADDEDGE(k, l), a traversal is performed following the sorted order of the vertex-list and the edge-lists. This ensures that addition of a new VNode or ENode does not violate the invariants of the data structure. The removal of a VNode or an ENode by a REMOVEVERTEX or REMOVEEDGE operation by default does not disturb the sorted order of the vertex-list or any edge-list. The lookup and GETPATH operations do not modify the data structure. Thus, it can be observed that the operations maintain the invariants of the data structure across their LPs.

This completes the proof of linearizability.　　□

4.2　Non-blocking Progress

THEOREM 4.2. In the presented algorithm
　(i) If the set of keys is finite, the operations CONTAINSVERTEX and CONTAINSEDGE are wait-free.
　(ii) The operation GETPATH is obstruction-free.
　(iii) The operations ADDVERTEX, REMOVEVERTEX, CONTAINSVERTEX, ADDEDGE, REMOVEEDGE, and CONTAINSEDGE are lock-free.

PROOF. It is easy to show (i). If the set of keys is finite, the graph size has a fixed upper bound. Which implies that there are only finite number of VNodes between $v(−∞)$ and $v(∞)$ in the vertex-list. A CONTAINSVERTEX(k) necessarily terminates on reaching $v(∞)$ which will be done in a finite number of steps of any non-faulty thread. A similar argument holds true for a CONTAINSEDGE.

It is also easy to see in the algorithm that whenever a modification operation is concurrent to a GETPATH, a COMPAREPATH or a COMPARETREE can not return true, enforcing the While loop at the line 219 in SCAN procedure to not terminate. Therefore, unless the steps are taken in isolation, a non-faulty thread that calls the GETPATH will not return as long as a concurrent operation is an arbitrary ADT operation. This shows (ii).

In the design of our algorithm, we can see that whenever an addition or a removal operation is obstructed by a concurrent removal operation by way of a marked pointer, the obstructing removal operation is necessarily helped ensuring its return. Addition and lookup do not require help by a concurrent operation. Therefore in an arbitrary concurrent execution comprising of any arbitrary data structure operation, at least one operation would complete in a finite number of steps taken by a non-faulty thread. Thus, the operations ADDVERTEX, REMOVEVERTEX, CONTAINSVERTEX, ADDEDGE, REMOVEEDGE, and CONTAINSEDGE are lock-free. This shows (iii).　　□

5　EXPERIMENTAL EVALUATION

We performed our tests on a workstation with Intel(R) Xeon(R) E5-2690 v4 CPU containing 56 cores running at 2.60GHz. Each core supports 2 logical threads. Every core's L1 - 64K, L2 - 256K cache memory is private to that core; L3-35840K cache is shared across the cores. The tests were performed in a controlled environment, where we were the sole users of the system. The implementation[b] has been done in C/C++ (without any garbage collection) and multi-threaded implementation is based on Posix threads.

In the experiments, we start with an initial graph of 1000 vertices and approximately $\binom{1000}{2}/4 ≈ 125000$ edges added randomly. The number of edges is approximately a quarter of the total number of edges in a complete graph with 1000 vertices. When the program begins, it creates a fixed number of threads (1, 10, 20, 30, 40, 50, 60 and 70) and each thread randomly performs a set of operations chosen by a particular workload distribution. The evaluation metric used is the number of operations completed in a unit time. We measure throughput obtained on running the experiment for 20 seconds. Each data point is obtained by averaging over 5 iterations. We compare the non-blocking graph with its sequential and coarse-grained counterparts in two separate sets of experiments comprising: (a) the ADT operations excluding GETPATH, and (b) all

---

[b]The complete source code of our implementation is available on Github [2].

the ADT operations. In the experiments, the following workload distributions were considered.
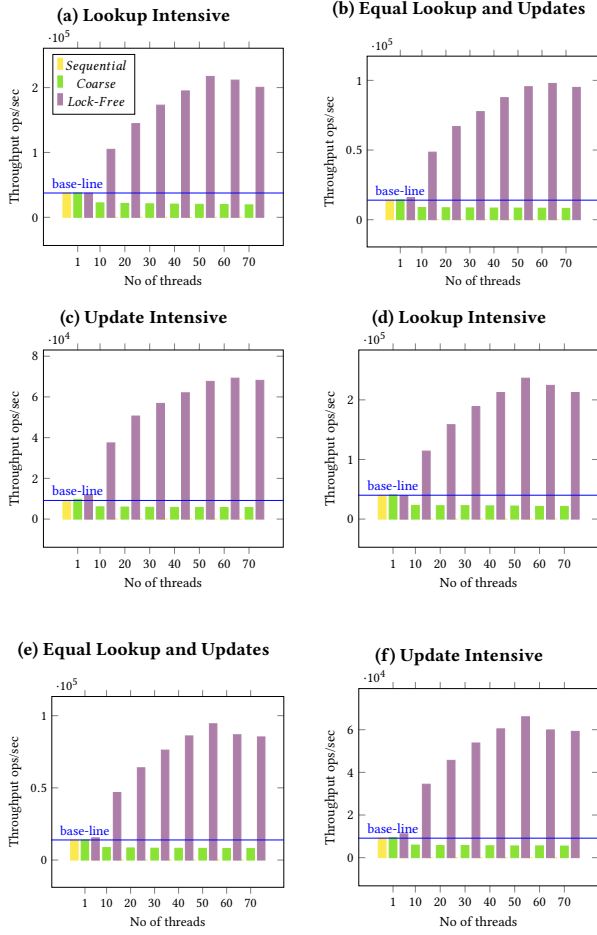

Figure 8: Concurrent Graph Data-Structure Results without and with GetPath. (a), (b) and (c) are without GetPath and (d), (e) and (f) are withGetPath.

In the first set of experiments, the distribution over the ordered set of operations {AddVertex, RemoveVertex, ContainsVertex, AddEdge, RemoveEdge, ContainsEdge } are (1) *Lookup Intensive*: (2.5%, 2.5%, 45%, 2.5%, 2.5%, 45%), see the Figure 8a. (2) *Equal Lookup and Updates*: (12.5%, 12.5%, 25%, 12.5%, 12.5%, 25%), see the Figure 8b. (3) *Update Intensive*: (22.5%, 22.5%, 5%, 22.5%, 22.5%, 5%), Figure 8c. In the second set of experiments, the distribution over the ordered set of operations {AddVertex, RemoveVertex, ContainsVertex, AddEdge, RemoveEdge, ContainsEdge, Get-Path } are - (1) *Lookup Intensive*: (2%, 2%, 45%, 2%, 2%, 45%, 2%), see the Figure 8d. (2) *Equal Lookup and Updates*: (24%, 24%, 12.5%, 24%, 24%, 12.5%, 2%), see the Figure 8e. (3) *Update Intensive*: (22.5%, 22.5%, 4%, 22.5%, 22.5%, 4%, 2%), see the Figure 8f. In this set of experiments, please note that we always take only 2% GetPath operations considering that its overhead in comparison to other operations is significant.

In the plots, firstly, we observe that the non-blocking algorithm is highly scalable with the number of threads in the system: only after

the available cores saturate with the threads, which is at 56 threads, the throughput numbers stop increasing. On the other hand, the coarse-grained lock-based version shows performance degradation with the increasing number of threads. This is in line with the serious drawback associated with coarse-grained locks: the contention among threads to acquire the global lock actually increases with the increasing number of threads. In fact, the coarse-grained lock-based implementation performs worse than even the sequential implementation, as soon as concurrency kicks in the system. In terms of absolute numbers, we see a marginal difference in the two sets of experiments. It indicates that the overhead of reachability query is light. On an average, we observe that the non-blocking algorithm offers 5-7x increase in the throughput in comparison to the sequential counterpart, which in any case outperforms the coarse-grained locking algorithm with multiple threads.

6 CONCLUSION

In this paper, we presented a non-blocking algorithm to implement a dynamic concurrent graph data structure, which allows threads to concurrently add and remove vertices/edges. The most important contribution of this work is an efficient obstruction-free reachability query in a concurrent graph. We prove the linearizability of the data structure operations. We extensively evaluated a sample C/C++ implementation of the algorithm through a number of micro-benchmarks. The non-blocking algorithm compared to a sequential and a coarse-grained lock-based concurrent version observably achieves up to 5-7x speedup with respect to their throughput.

There are several graph databases that consider dynamic graph operations. But we have not considered them in this work as none of them, to the best of our knowledge, are non-blocking nor do they satisfy linearizability.

REFERENCES

[1] Yehuda Afek, Hagit Attiya, Danny Dolev, Eli Gafni, Michael Merritt, and Nir Shavit. 1993. Atomic Snapshots of Shared Memory. *J. ACM* 40, 4 (1993), 873–890.

[2] Bapi Chatterjee, Sathya Peri, Muktikanta Sa, and Nandini Singhal. 2017. ConcurrentGraphDS. https://github.com/PDCRL/ConcurrentGraphDS.

[3] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. 2009. *Introduction to algorithms*. MIT press.

[4] Nhan Nguyen Dang and Philippas Tsigas. 2011. Progress Guarantees when Composing Lock-free Objects. In *Proceedings of the 17th International Conference on Parallel Processing - Volume Part II (Euro-Par'11)*. 148–159.

[5] Timothy L. Harris. 2001. A Pragmatic Implementation of Non-blocking Linked-Lists. In *Distributed Computing, 15th International Conference, DISC 2001, Lisbon, Portugal, October 3-5, 2001, Proceedings*. 300–314.

[6] Maurice Herlihy, Victor Luchangco, and Mark Moir. 2003. Obstruction-Free Synchronization: Double-Ended Queues as an Example. In *23rd International Conference on Distributed Computing Systems (ICDCS 2003), 19-22 May 2003, Providence, RI, USA*. 522–529.

[7] Maurice Herlihy and Nir Shavit. 2012. *The Art of Multiprocessor Programming, Revised Reprint* (1st ed.). Morgan Kaufmann Publishers Inc.

[8] Maurice P. Herlihy and Jeannette M. Wing. 1990. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.* 12, 3 (1990), 463–492.

[9] Nikolaos D. Kallimanis and Eleni Kanellou. 2015. Wait-Free Concurrent Graph Objects with Dynamic Traversals. In *19th International Conference on Principles of Distributed Systems, OPODIS 2015, December 14-17, 2015, Rennes, France*. 27:1–27:17.