



A Pragmatic Non-blocking Concurrent Directed Acyclic Graph

Sathya Peri^{1(✉)}, Muktikanta Sa^{1(✉)}, and Nandini Singhal^{2(✉)}

¹ Department of Computer Science and Engineering,
Indian Institute of Technology Hyderabad, Sangareddy, India
{sathya_p,cs15resch11012}@iith.ac.in

² Microsoft (R&D) Pvt. Ltd., Bangalore, India
nandini12396@gmail.com

Abstract. In this paper, we have developed two non-blocking algorithms for maintaining acyclicity in a concurrent directed graph. The first algorithm is based on a *wait-free reachability* query and the second one on *partial snapshot-based obstruction-free reachability* query. Interestingly, we are able to achieve the acyclic property in a dynamic setting without (1) making use of helping descriptors by other threads, or (2) clean double collect mechanism. We present a proof to show that the graph remains acyclic at all times in the concurrent setting. We also prove that the acyclic graph data-structure operations are linearizable. We implement both the algorithms in C++ and test through several micro-benchmarks. Our experimental results illustrate an average of 7x improvement over the sequential and global-lock implementation.

Keywords: Acyclic graph · Concurrent data structure · Linearizability · Lock-freedom

1 Introduction

A graph is a common data-structure that can model many real-world objects and pairwise relationships among them. Graphs have a huge number of applications in various fields like social networking, VLSI design, road networks, graphics, blockchains and many more. Usually, these graphs are *dynamic* in nature, that is, they undergo dynamic changes like addition and removal of vertices and/or edges [9]. These applications also need data-structure which supports dynamic changes and can expand at run-time depending on the availability of memory in the machine.

Nowadays, multi-core systems have become ubiquitous. To fully harness the computational power of these systems, it has become necessary to design efficient data-structures which can be executed by multiple threads concurrently. In the

This work is partly funded by a research grant from Intel, USA.

N. Singhal—Work done while a student at IITH.

© Springer Nature Switzerland AG 2019

M. F. Atig and A. A. Schwarzmann (Eds.): NETYS 2019, LNCS 11704, pp. 327–344, 2019.

https://doi.org/10.1007/978-3-030-31277-0_22

past decade, there have been several efforts to port sequential data-structures to a concurrent setting, like stacks [12], queues [2, 16], sets [10, 11, 17, 23], trees [5, 19].

Most of these data-structure use locks to handle mutual exclusion while doing any concurrent modifications. However, in an asynchronous shared-memory system, where an arbitrary delay or a crash failure of a thread is possible, a lock-based implementation is vulnerable to arbitrary delays or deadlock. For instance, a thread could acquire a lock and then sleep (or get swapped out) for a long time, or the thread could get involved in a deadlock with the other threads while obtaining locks, or even crash after obtaining a lock.

On the other hand, in a lock-free data-structure, threads do not acquire locks. Instead, they use atomic hardware instructions such as compare-and-swap, test-and-set etc. These instructions ensure that at least one non-faulty thread is guaranteed to finish its operation in a finite number of steps. Therefore, lock-free data-structures are highly scalable and naturally fault-tolerant.

Although several concurrent data-structures have been developed, concurrent graph data-structures and the related operations are still largely unexplored. In several graph applications, one of the crucial requirements is preserving *acyclicity*. Acyclic graphs are often applied to problems related to databases, data processing, scheduling, finding the best route during navigation, data compression, blockchains etc. Applications relying on graphs mostly use a sequential implementation and the accesses to the shared data-structures are synchronized through the global-locks, which causes serious performance bottlenecks.

A relevant application is *Serialization Graph Testing (SGT)* in Databases [24, Chap 4] and Transactional Memory (TM) [22]. SGT requires maintaining an acyclic graph on all concurrently executing (database or TM) transactions with edges between the nodes representing conflicts among them. In a concurrent scenario, where multiple threads perform different operations, maintaining acyclicity without using locks is not a trivial task. Indeed, it requires every shared memory access to be checked for the violation of the acyclic property, which necessitates that all the operations be efficient.

Apart from SGT, several popular blockchains maintain acyclic graphs such as tree structure (Bitcoin [3, 18], Ethereum [4] etc.) or general DAGs (Tangle [21]).

1.1 Contributions

In this paper, we present an efficient non-blocking concurrent acyclic directed graph data-structure. Its operations are similar to the concurrent graph proposed by Chatterjee et al. [6] with some non-trivial modifications. The contributions of our work are summarized below:

1. We describe an Abstract Data Type (ADT) that maintains an acyclic directed graph $G = (V, E)$. It comprises of the following methods on the sets V and E : (1) Add Vertex: ACYADDV (2) Remove Vertex: ACYREMV, (3) Contains Vertex: ACYCONV (4) Add Edge: ACYADDE (5) Remove Edge: ACYREME

- and (6) Contains Edge: `ACYCONE`. The ADT remains acyclic after completion of any of the above operations in G . The acyclic graph is represented as an adjacency list.
2. We present an efficient concurrent non-blocking implementation of the ADT (Sect. 3). We present two approaches for maintaining acyclicity: the first one is based on a wait-free reachability query (`SCR`: Single Collect Reachable) and the second one is based on obstruction-free reachability query (`DCR`: Double Collect Reachable) similar to the `GETPATH` method of Chatterjee et al. [6] (Sect. 4).
 3. We prove the correctness by showing the operations of the concurrent acyclic graph data-structure are linearizable [14]. We also prove the non-blocking progress guarantee, specifically we prove: (a) The operations `ACYCONV` and `ACYCONE` are wait-free, only if the vertex keys are finite; (b) Among the two algorithms for maintaining acyclicity, we show that the first algorithm based on searchability is wait-free, whereas the second algorithm based on reachability queries is obstruction-free and (c) The operations `ACYADDV`, `ACYREMV`, `ACYCONV`, `ACYADDE`, `ACYREME`, and `ACYCONE` are lock-free Sect. 5.
 4. We implemented the non-blocking algorithms in C++ and evaluated over a number of micro-benchmarks. Our experimental results depict on an average of 7x improvement over the sequential and global lock implementation (Sect. 6).

1.2 Related Work

Kallimanis and Kanellou [15] presented a concurrent graph that supports wait-free edge updates and traversals. They use an adjacency matrix representation for the graph, with a bounded number of vertices. As a result, their data-structure does not allow any insertion or deletion of vertices after initialization of the graph. This may not be adequate for many real-world applications which need dynamic modifications of vertices as well as unbounded graph size.

A recent work by Chatterjee et al. [6] proposed a non-blocking concurrent graph data-structure which allows multiple threads to perform dynamic insertion and deletion of vertices & edges. Our paper extends this data-structure to maintain acyclicity of a directed graph.

1.3 Overview of the Algorithm Design

Before getting into the technical details (in Sect. 3) of the algorithm, we first provide an overview of the design. We implement an acyclic concurrent unbounded directed graph as a concurrent list of linked lists [11] also used by Chatterjee et al. [6]. The *vertex-nodes* are placed in a sorted linked-list and the neighboring vertices of each vertex-node are placed in a rooted sorted linked-list of *edge-nodes*. To achieve efficient graph traversal, we maintain a pointer from each edge-node to its corresponding vertex-node. Each vertex-node's edge-list and vertex-list are lock-free with respect to concurrent update and lookup operations.

As we know that lock-freedom is not composable [8] and our algorithm is a composition of lock-free operations, we prove the liveness of our algorithm independent of the lock-free list arguments. In addition to that, we also propose some refined optimizations for the concurrent acyclic graph operations that not only enhance the performance but also simplify the design.

Our main requirement is preserving *acyclicity* and one can see that a cycle is created only after inserting an edge to the graph. So, after the insertion of a new edge to the graph, we verify if the resulting graph is acyclic or not. If it creates a cycle, we simply delete the inserted edge from the graph. However, the challenge is that these intermediate steps must be oblivious to the user and the graph must always appear to be acyclic. We ensure this by adding a *transit* field to the edges that are temporarily added. To verify the acyclic property of the graph, we propose two efficient algorithms: first one based on a wait-free reachability query and the second one based on obstruction-free reachability query similar to the GET-PATH operation of [6]. Both the reachability algorithms perform breadth-first search (BFS) traversal. For the sake of efficiency, we implement BFS traversal in a non-recursive manner. However, in order to achieve overall performance, we do not make use of *helping descriptors* for the reachability queries.

2 System Model and Preliminaries

The Memory Model. We consider an asynchronous shared-memory model with a finite set of p processors accessed by a finite set of n threads. The non-faulty threads communicate with each other by invoking methods on the shared objects. We run our acyclic graph data-structure on a shared-memory multi-core system with multi-threading enabled which supports atomic `read`, `write`, `fetch-and-add` (FAA) and `compare-and-swap` (CAS) instructions.

Correctness. We consider *linearizability* proposed by Herlihy and Wing [14] as the correctness criterion for the graph operations. We assume that the execution generated by a data-structure is a collection of method invocation and response events. Each invocation of a method call has a subsequent response. An execution is linearizable if it is possible to assign an atomic event as a *linearization point* (LP) inside the execution interval of each method such that the result of each of these methods is the same as it would be in a sequential execution in which the methods are ordered by their LPs [14].

Progress. The *progress* properties specify when a thread invoking operations on the shared memory objects completes in the presence of other concurrent threads. In this context, we present an acyclic graph implementation with operations that satisfies *lock-freedom*, based on the definitions in Herlihy and Shavit [13].

3 The Data Structure

3.1 Abstract Data Type

An acyclic graph is defined as a directed graph $G = (V, E)$, where V is the set of vertices and E is the set of directed edges. Each edge in E is an ordered pair of vertices belonging to V . Every vertex has an immutable unique key. The vertex represented by the key k is denoted k . A directed edge from the vertex k to l is denoted as $e(k, l) \in E$.

For a concurrent acyclic graph, we define following ADT operations:

1. **ACYADDV**(k) adds a vertex k to V , only if $k \notin V$ and then returns **true**, otherwise it returns **false**.
2. **ACYREMOV**(k) deletes a vertex k from V , only if $k \in V$ and then returns **true**, otherwise it returns **false**. Once a vertex k is deleted successfully, all its outgoing and incoming edges are also removed.
3. **ACYCONV**(k) returns **true** only if $k \in V$, otherwise it returns **false**.
4. **ACYADDE**(k, l) operation is slightly involved and works as follows.
 - (a) It adds an edge $e(k, l)$ to E , if (i) $k \in V$ and $l \in V$ (ii) $e(k, l) \notin E$ and adding it does not create a cycle in the graph. If either of the conditions (i) or (ii) are not satisfied, the edge is not added to E and it returns **false** along with an indicative strings **VERTEX NOT PRESENT**, **EDGE ALREADY PRESENT** or **CYCLE DETECTED** depending on execution.
 - (b) If both (i) and (ii) conditions mentioned above are true and there is no concurrent edge addition, then this method adds the edge $e(k, l)$ to E and returns **true** along with an indicative string **EDGE ADDED**.
 - (c) If both (i) and (ii) conditions, mentioned in Step 4a, are true and there is a concurrent edge addition (such as $e(u, v)$) then the edge $e(k, l)$ may or may not get added to E . In case, $e(k, l)$ gets added to E , then the method returns **true** along with an indicative string **EDGE ADDED**. Otherwise, it returns **false** along with **CYCLE DETECTED**.

There is an inherent non-determinism in this edge addition procedure. It can be seen from Step 4c that this method may return **false** in presence of other concurrent edge additions. But if the primary requirement is to ensure that the graph remains acyclic such as in SGT or blockchains, then this behaviour is acceptable.

5. **ACYREME**(k, l) deletes the edge $e(k, l)$ from E , only if $e(k, l) \in E$ and $k \in V$ and $l \in V$ then it returns **true** along with an indicative string **EDGE REMOVED**. If $k \notin V$ or $l \notin V$, it returns **false** along with a string **VERTEX NOT PRESENT**. If $e(k, l) \notin E$, it returns **false** along with a string **EDGE NOT PRESENT**.
6. **ACYCONE**(k, l) if $e(k, l) \in E$ and $k \in V$ and $l \in V$ then it returns **true** along with a string **EDGE PRESENT**, otherwise it returns **false** along with a string **VERTEX OR EDGE NOT PRESENT**.

3.2 The Data-Structures

The algorithm uses three kinds of nodes structures: **VNode**, **ENode** and **BFSNode**. These structures and the adjacency list representation of an acyclic graph are

shown in Fig. 1. The `VNode` structure has five fields, two pointers `vnnext` and `enext`, an immutable key `k`, an atomic counter `ecount`, and a `VisitedArray` array. The use of `ecount` and `VisitedArray` are described in the later section. The pointer `vnnext` is an atomic pointer pointing to the next `VNode` in the vertex-list, whereas, an `enext` pointer points to the edge head of the edge-list of a `VNode`. Similarly, an `ENode` structure has three fields, two pointers `enext` and `pointv` and an immutable key `l`. The `enext` is an atomic pointer pointing to the next `ENode` in the edge-list and `pointv` points to the corresponding `VNode`, which helps direct access to its `VNode` while performing any traversal like BFS, DFS, etc. We assume that all the `VNodes` have a unique identification key `k` and all the adjacency `ENodes` of a `VNode` have also a unique key `l`.

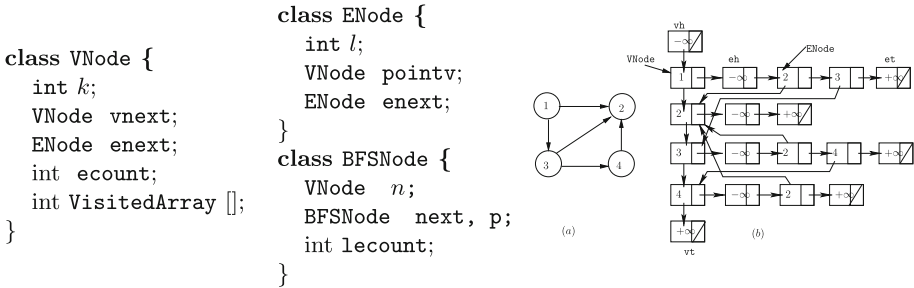


Fig. 1. Node structures used in the acyclic graph data-structure: `ENode`, `VNode` and `BFSNode`. (a) An acyclic graph (b) The concurrent acyclic graph representation of data-structure for (a).

A `BFSNode` has three pointers `n`, `next` and `p`, and a counter `lecount`. The pointer `n` holds the corresponding `VNode`'s address, `next` points to the next `BFSNode` in the BFS-list and `p` points to the corresponding parent. The local counter `lecount` stores `n`'s `ecount` value which is used in the `COMPARETREE` and `COMPAREPATH` methods.

We initialize the vertex-list with dummy head(`vh`) and tail(`vt`) (called sentinels) with values $-\infty$ and ∞ respectively. Similarly, each edge-lists is also initialized with dummy head (`eh`) and tail (`et`) (refer Fig. 1).

Our acyclic graph data-structure maintains some *invariants*: (a) the vertex-list is sorted based on the `VNode`'s key value `k` and each unmarked `VNode` is reachable from `vh`, (b) also each of the edge-lists are sorted based on the `ENode`'s key value `l` and unmarked `ENodes` are reachable from `eh` of the corresponding `VNode` and (c) the concurrent graph always stays *acyclic*.

4 Working of the Non-blocking Algorithm

In this section, we describe the technical details of all the acyclic graph operations.

Pseudo-code Convention: The acyclic graph algorithm is depicted in Figs. 2, 3, 4, 6. We use $p.x$ to access the member field x of a class object pointer p . To return multiple variables from an operation, we use $\langle x_1, x_2, \dots, x_n \rangle$. To avoid the overhead of another field in the node structure, we use bit-manipulation: we use last two significant bits of a pointer p . We define six methods that manipulate these bits: (a) $\text{ISMARKED}(p)$ and $\text{ISTRANSIT}(p)$, return **true** if the last two significant bits of pointer p are set to 01 and 10, respectively, else, both return **false**, (b) $\text{MARKEDREF}(p)$, $\text{UNMARKEDREF}(p)$, $\text{ADDEDREF}(p)$ and $\text{TRANSITREF}(p)$ sets the last two significant bits of the pointer p to 01, 00, 11 and 10, respectively. An invocation of $\text{ACYCVNODE}(k)$ creates a new **VNode** with key k . Similarly, an invocation of $\text{ACYCENODE}(k)$ creates a new **ENode** with key k in **TRANSIT** state (explained below). Whereas, an invocation of $\text{ACYCBNODE}(k)$ creates a new **BFSNode** with vertex k . For a newly created **VNode**, the pointer fields are **NULL**. Similarly, a newly created **ENode** initialises its pointer fields to **NULL** as well. In case of a new **BFSNode**, the pointer field n , next and p are initialized with k , **NULL** and parent node, respectively. Each slot of a **VisitedArray** in each **VNode** is initialized to 0 and the counter ecount is also initialized to 0.

To ensure acyclicity, we use a operation descriptor with a pointer in a single memory-word with *bit-masking*. In case of an x86-64 bit architecture, memory has a 64-bit boundary and the last three least significant bits are unused. So, our operator descriptor uses the last two significant bit of the pointer. If the last two bits are set to: (a) 01 then the pointer is **MARKED**, (b) 10 indicates the pointer is in **TRANSIT**, (c) 11 value of the pointer indicates **ADDED** and (d) 00 indicates the pointer is unused and unmarked.

We next describe the vertex and edge operations. We use the term method and operation interchangeably in the rest of this document.

4.1 Acyclic Vertex Operations

The acyclic vertex operations ACYADDV , ACYREMV and ACYCONV are depicted in Fig. 2. The ACYCONV method does not help other threads in the process of traversal from the vertex head vh to the destination vertex. If the keys in the vertex set are finite, then the ACYCONV operation is wait-free.

An $\text{ACYADDV}(key)$ operation is invoked by passing the key to be inserted, in Lines 1 to 14. It first traverses the vertex-list in a lock-free manner starting from vh using LOCV procedure (Line 3) until it finds a vertex with its key greater than or equal to key . In the process of traversal, it physically deletes all logically deleted **VNodes** using **CAS** operation for helping previously pending ACYREMV operations. Once it reaches the appropriate location, say currv and has identified its predecessor, say predv , it checks if the key is already present. If the key is not present, it attempts to add the new **VNode**, say newv in between the predv and currv (Line 9) using **CAS** operation. If the **CAS** is unsuccessful, then these steps are retried. On the other hand, if key is already present then the method returns **false**.

Like an ACYADDV , an $\text{ACYREMV}(key)$ operation is invoked by passing the key to be deleted, in Lines 15 to 31. It traverses the vertex-list in a lock-free

manner starting from vh using LOCV procedure (Line 17) until it finds a vertex with its key greater than or equal to key . Similar to the ACYADDV, during the traversal it physically deletes all logically removed VNodes using CAS operations for helping other pending ACYREMOV operations. Once it reaches the appropriate location, say $currv$ and its predecessor, say $predv$, it checks to see if key is already present. If present, it attempts to remove $currv$ in two steps (like [11]), (a) atomically marking the $vnext$ of $currv$ using a CAS (Line 23), and (b) atomically updating the $vnext$ of the $predv$ to point to the $vnext$ of $currv$ using a CAS (Line 24). On any unsuccessful CAS, these steps are reattempted. If the key is not present then, this method returns **false**.

```

1: Operation ACYADDV( $key$ )
2: while (1) do
3:    $\langle predv, currv \rangle \leftarrow \text{LOCV}(vh, key)$ ;
4:   if ( $currv.k = key$ ) then
5:     return false;
6:   else
7:      $newv \leftarrow \text{ACYCVNODE}(key)$ ;
8:      $newv.vnext \leftarrow currv$ ;
9:     if (CAS ( $predv.vnext, currv, newv$ )) then
10:       return true;
11:     end if
12:   end if
13: end while
14: end Operation


---


15: Operation ACYREMOV( $key$ )
16: while (1) do
17:    $\langle predv, currv \rangle \leftarrow \text{LOCV}(vh, key)$ ;
18:   if ( $currv.k \neq key$ ) then
19:     return false;
20:   end if
21:    $cnext \leftarrow currv.vnext$ ;
22:   if ( $\neg \text{ISMARKED}(cnext)$ ) then
23:     if (CAS ( $currv.vnext, cnext, \text{MARKEDREF}$ 
24: ( $cnext$ ))) then
25:       if (CAS ( $predv.vnext, currv, cnext$ ))
26:         then
27:            $break$ ;
28:         end if
29:       end if
30:     end if
31:   end while
32: return true;
33: end Operation


---


32: Operation ACYCONV( $key$ )
33:  $currv \leftarrow vh.vnext$ ;
34: while ( $currv.k < key$ ) do
35:    $currv \leftarrow \text{UNMARKEDREF}$ 
36: ( $currv.vnext$ );
37: end while
38: if ( $currv.k = key \wedge \neg \text{ISMARKED}$ 
39: ( $currv$ )) then
40:   return true;
41: else
42:   return false;
43: end if
44: end Operation


---


43: Operation ACYCONE( $k, l$ )
44:  $\langle u, v, st \rangle \leftarrow \text{CONCPLUS}(k, l)$ ;
45: if ( $st = \text{false}$ ) then
46:   return  $\langle \text{false}, \text{"VERTEX NOT PRESENT"} \rangle$ ;
47: end if
48:  $curre \leftarrow u.enext$ ;
49: while ( $curre.l < l$ ) do
50:    $curre \leftarrow \text{UNMARKEDREF}$ 
51: ( $curre.enext$ );
52: end while
53: if ( $curre.l = l \wedge \neg \text{ISMARKED}(u) \wedge \neg$ 
54:  $\text{ISMARKED}(v) \wedge \neg \text{ISMARKED}(curre) \wedge \neg$ 
55:  $\text{ISTRANSIT}(curre)$ ) then
56:   return  $\langle \text{true}, \text{"EDGE PRESENT"} \rangle$ ;
57: else
58:   return  $\langle \text{false}, \text{"VERTEX OR EDGE NOT$ 
59:  $\text{PRESENT"} \rangle$ ;
60: end if
61: end Operation

```

Fig. 2. Pseudo-codes of ACYADDV, ACYREMOV, ACYCONV and ACYCONE

When a vertex is deleted from a graph, all its incoming and outgoing edges should also get removed. Once a CAS at Line 23 is successful, the vertex is logically deleted from the vertex-list and its outgoing edges are deleted atomically. Notice that, all the incoming edges are logically deleted from the corresponding ENodes of any edge-lists. This is because each ENode has a direct pointer $pointv$ to its vertex node and calls ISMARKED to validate the deleted VNode. Finally, these ENodes are physically deleted using CAS operation by any other helping edge operation (which is described later).

An $\text{ACYCONV}(key)$ operation, first traverses the vertex-list in a wait-free manner skipping all the logically marked VNodes until it finds a vertex with its key greater than or equal to key (in Lines 32 to 42). Once it reaches the appropriate VNode , it checks if its key value is equal to the key and if it is unmarked, then it returns **true** otherwise returns **false**. ACYCONV method does not help other threads during the traversal.

4.2 Acyclic Edge Operations

The acyclic edge operations ACYADDE and ACYREME are depicted in Fig. 3 and ACYCONE is depicted in Fig. 2.

```

58: Operation  $\text{ACYADDE}(k, l)$ 
59:    $\langle u, v, st \rangle \leftarrow \text{ACYCONVPLUS}(k, l)$ ;
60:   if ( $st = \text{false}$ ) then
61:     return  $\langle \text{false}, \text{VERTEX NOT PRESENT} \rangle$  ;
62:   end if
63:   while (1) do
64:     if ( $\text{ISMARKED}(u) \vee \text{ISMARKED}(v)$ ) then
65:       return  $\langle \text{false}, \text{VERTEX NOT PRESENT} \rangle$  ;
66:     end if
67:      $\langle \text{prede}, \text{curre} \rangle \leftarrow \text{LOCE}(u.\text{enext}, l)$ ;
68:     if ( $\text{curre}.l = l$ ) then
69:       return  $\langle \text{false}, \text{EDGE ALREADY PRESENT} \rangle$  ;
70:     end if
71:      $\text{newe} \leftarrow \text{ACYCENODE}(l)$ ;
72:      $\text{newe}.\text{enext} \leftarrow \text{TRANSITREF}(\text{curre})$ ;
73:      $\text{newe}.\text{pointv} \leftarrow v$ ;
74:      $\text{nnxt} \leftarrow \text{newe}.\text{enext}$ ;
75:     if ( $\text{CAS}(\text{prede}.\text{enext}, \text{curre}, \text{newe})$ ) then
76:       if ( $\neg \text{SCR}(v, u)$ ) then // SCR or DCR is
invoked
77:          $\text{newe}.\text{enext} \leftarrow \text{ADDEDREF}(\text{nnxt})$ ;
78:          $u.\text{ccount}.\text{FetchAndAdd}(1)$ ; // only if
DCR is invoked
79:         return  $\langle \text{true}, \text{EDGE ADDED} \rangle$  ;
80:       else
81:          $\text{newe}.\text{enext} \leftarrow \text{MARKEDREF}(\text{nnxt})$ ;
82:         return  $\langle \text{false}, \text{CYCLE DETECTED} \rangle$  ;
83:       end if
84:     end if
85:   end while
86: end Operation

87: Operation  $\text{ACYREME}(k, l)$ 
88:    $\langle u, v, st \rangle \leftarrow \text{ACYCONVPLUS}(k, l)$ ;
89:   if ( $st = \text{false}$ ) then
90:     return  $\langle \text{false}, \text{"VERTEX NOT PRESENT"} \rangle$  ;
91:   end if
92:   while (1) do
93:     if ( $\text{ISMARKED}(u) \vee \text{ISMARKED}(v)$ ) then
94:       return  $\langle \text{false}, \text{"VERTEX NOT PRESENT"} \rangle$ ;
95:     end if
96:      $\langle \text{prede}, \text{curre} \rangle \leftarrow \text{LOCE}(u.\text{enext}, l)$ ;
97:     if ( $\text{curre}.l \neq l$ ) then
98:       return  $\langle \text{false}, \text{"EDGE NOT PRESENT"} \rangle$ ;
99:     end if
100:     $\text{cnt} \leftarrow \text{curre}.\text{enext}$ ;
101:    if ( $\neg \text{ISMARKED}(\text{cnt})$ ) then
102:      if ( $\text{CAS}(\text{curre}.\text{enext}, \text{cnt}, \text{MARKEDREF}$ 
 $(\text{cnt}))$ ) then
103:         $u.\text{ccount}.\text{FetchAndAdd}(1)$ ; // only
if DCR is invoked
104:        if ( $\text{CAS}(\text{prede}.\text{enext}, \text{curre}, \text{cnt})$ )
then
105:          break;
106:        end if
107:      end if
108:    end if
109:    end while
110:    return  $\langle \text{true}, \text{"EDGE REMOVED"} \rangle$ ;
111: end Operation

```

Fig. 3. Pseudo-codes of ACYADDE and ACYREME .

An $\text{ACYADDE}(k, l)$ operation, begins in Lines 58 to 86 by validating the presence of the k and l in the vertex-list by invoking ACYCONVPLUS (Line 59) and validating that both the vertices are unmarked (Line 64). If the validations fail, it returns **false** along with an indicative string **VERTEX NOT PRESENT**. Once the validation succeeds, LOCE is invoked (Line 67) to find the location to insert $e(k, l)$ in the edge-list of the k . The operation LOCE works similar to the helping method LOCV ; except that in the traversal phase, it physically deletes two kinds of logically deleted ENodes (to help a pending incompleting ACYADDE or ACYREME operations): (a) ENodes whose VNode has already been logically

deleted using a CAS, and (b) the logically deleted ENodes using a CAS. The operation LOCE traverses the edge-list until it finds an ENode with its key greater than or equal to l . Once it reaches the appropriate location, say *curre* and its predecessor, say *prede*, it checks if the key l is already present. If the key is already present, it simply returns **false** along with an indicative string **EDGE ALREADY PRESENT**. Otherwise, it attempts a CAS to add a new $e(k, l)$ with **TRANSIT** state in between *prede* and *curre* (Line 75). On an unsuccessful CAS, the operation is re-tried.

Once the edge $e(k, l)$ is inserted in a transit state, it invokes the reachability method to test whether this edge has created a cycle. As explained earlier, this method returns false if adding this edge creates a cycle. Further, the reachability method can return false even if this edge does not create a cycle in presence of other concurrent ACYADDE methods.

As mentioned earlier, we have proposed two algorithms to maintain the acyclicity property. First one is the wait-free reachable algorithm SCR, and the second one is the obstruction-free reachable algorithm DCR. The detailed working of these algorithms is given in the subsequent subsections. If the edge $e(k, l)$ creates a cycle, we delete it by setting its state from **TRANSIT** to **MARKED** (Line 81) and return **false** along with an indicative string **CYCLE DETECTED**. Otherwise, we set the state from **TRANSIT** to **ADDED** (Line 77) and return **true** along with an indicative string **EDGE ADDED**. Like ACYADDE, an ACYREME(k, l) operation (Lines 87 to 111), first validates the presence of the corresponding VNodes and check if they are unmarked. If the validations fail, it returns **false** along with an indicative string **VERTEX NOT PRESENT**. Once the validation succeeds, it finds the location to delete the $e(k, l)$ in the edge-list of the k . Similar to ACYADDE, in the traversal phase, it also physically deletes two kinds of logically deleted ENodes: (a) ENodes whose VNode has been logically deleted, and (b) the logically deleted ENodes. It traverses the edge-list until it finds an ENode with its key greater than or equal to l . Once it reaches the appropriate location, say *curre* and its predecessor, say *prede*, it checks if the key l is already present. If the key is not present, it returns **false** along with a string **EDGE NOT PRESENT**; otherwise it attempts to remove *curre* in two steps: (a) atomically marking the **enext** of *curre* using a CAS (Line 102), and then (b) atomically updating the **enext** of *prede* to point to the **enext** of *curre* using a CAS (Line 104). On any unsuccessful CAS, it reattempts this process. After a successful CAS, it returns **true** along with a string **EDGE REMOVED**.

Similarly, an ACYCONE(k, l) operation, in Lines 43 to 57, validates the presence of the corresponding VNodes. Then it traverses the edge-list of k in a wait-free manner skipping all logically marked ENodes until it finds an edge with its key greater than or equal to l . Once it reaches the appropriate ENode, checks its key value equal to l and it is unmarked and not in **TRANSIT** state and also k and l are unmarked, then it returns **true** along with a string **EDGE PRESENT** otherwise it returns **false** along with a sting **VERTEX OR EDGE NOT PRESENT**. Like ACYCONV, we also do not allow ACYCONE for any helping thread in the process of traversal.

4.3 Wait-Free Single Collect Reachable Algorithm

In this subsection, we describe one of our algorithms to detect the cycle of a concurrent graph in a wait-free manner. As mentioned earlier, a cycle can be only be formed on adding an edge to the graph. The SCR (k, l) operation, in Lines 112 to 137, performs non-recursive BFS traversal starting from the vertex k . Reader can refer [7] to know the working of the BFS traversals in graphs. In the process of BFS traversal, it explores `VNodes` which are reachable from k and unmarked. If it reaches l , then it terminates by returning `true` to the `ACYADDE` operation. Then `ACYADDE` deletes $e(k, l)$ by setting `enext` pointer from the `TRANSIT` state to `MARKED` state and returns `false` along with an indicative string `CYCLE DETECTED`. If it is unable to reach l from k after exploring all reachable `VNodes` through `TRANSIT` or `ADDED` or unmarked `ENodes`, then it terminates by returning `false` to the `ACYADDE` operation. Now `ACYADDE` adds $e(l)$ by setting `enext` pointer from the `TRANSIT` state to `ADDED` state and then it returns `true` along with an indicative string `EDGE ADDED`, which preserves the acyclic property after `ACYADDE` (Figs. 4 and 5).

In the process of BFS traversal, we have used a `VisitedArray` (with size as that of the number of threads) to put all the visited `VNodes` locally. This is because multiple threads repeatedly invoke reachable operation concurrently, a boolean variable or a boolean array would not suffice like in case of sequential execution. We have used a thread local variable `cnt` as a counter for the number

```

112: Operation SCR ( $k, l$ )
113:   tid  $\leftarrow$  this_thread.get_id();
114:   queue  $\langle$  VNode  $\rangle$  Q;
115:   cnt  $\leftarrow$  cnt + 1;
116:   k.visitedArray[tid]  $\leftarrow$  cnt ;
117:   Q.enqueue( $k$ );
118:   while ( $\neg$ Q.empty()) do
119:     VNode cvn  $\leftarrow$  Q.dequeue();
120:     eh  $\leftarrow$  cvn.enext;
121:     for (ENode itn  $\leftarrow$  eh.enext to et) do
122:       if ( $\neg$ ISMARKED (itn)) then
123:         VNode adjn  $\leftarrow$  itn.pointv;
124:         if ( $\neg$ ISMARKED (adjn)) then
125:           if (adjn =  $l$ ) then
126:             return true;
127:           end if
128:           if (adjn[tid]  $\neq$  cnt) then
129:             adjn.VisitedArray [tid]  $\leftarrow$  cnt;
130:             Q.enqueue(adjn);
131:           end if
132:         end if
133:       end if
134:     end for
135:   end while
136:   return false;
137: end Operation

138: Operation BFSTREECOLLECT ( $k, l$ )
139:   queue  $\langle$  BFSNode  $\rangle$  Q; cnt  $\leftarrow$  cnt + 1;
140:   k.visitedArray[tid]  $\leftarrow$  cnt ;
141:   bNode  $\leftarrow$  ACYCBNODE ( $k, \text{NULL}, \text{NULL},$ 
    k.ecount);
142:   bTree.Insert(bNode); Q.enqueue(bNode);
143:   while ( $\neg$ Q.empty()) do
144:     BFSNode cvn  $\leftarrow$  Q.dequeue();
145:     eh  $\leftarrow$  cvn.n.enext;
146:     for (ENode itn  $\leftarrow$  eh.enext to et) do
147:       if ( $\neg$ ISMARKED (itn)) then
148:         VNode adjn  $\leftarrow$  itn.pointv;
149:         if ( $\neg$ ISMARKED (adjn)) then
150:           if (adjn =  $l$ ) then
151:             bNode  $\leftarrow$  ACYCBNODE (adjn, cvn,
    NULL, adjn.ecount);
152:             bTree.Insert(bNode);
153:             return  $\langle$  bTree, true  $\rangle$ ;
154:           end if
155:           if (adjn[tid]  $\neq$  cnt) then
156:             adjn.VisitedArray [tid]  $\leftarrow$  cnt;
157:             bNode  $\leftarrow$  ACYCBNODE (adjn, cvn,
    NULL, adjn.ecount);
158:             bTree.Insert(bNode);
159:             Q.enqueue(bNode);
160:           end if
161:         end if
162:       end if
163:     end for
164:   end while
165:   return  $\langle$  bTree, false  $\rangle$ ;
166: end Operation

```

Fig. 4. Pseudo-codes of SCR and BFSTREECOLLECT.

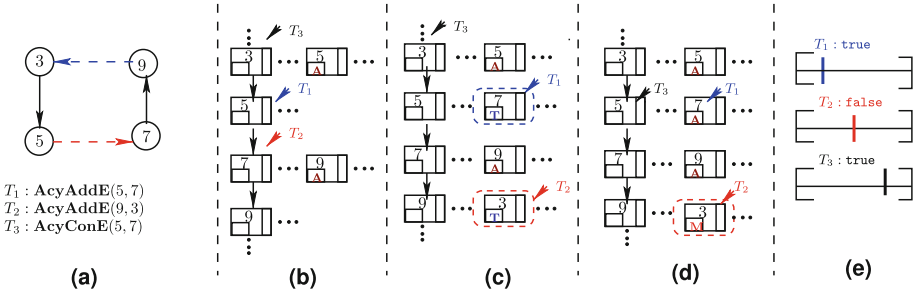


Fig. 5. An example working of the methods while preserving acyclicity. (a) The initial graph, T_1 , T_2 and T_3 are concurrently performing operations. The corresponding data-structure is shown in (b). In (c), T_3 is traversing the vertex list, while T_1 and T_2 have added their corresponding edges in **TRANSIT**, T state and performing cycle detection. (d) T_1 has succeeded; and changed the status to **ADDED**, A . However, T_2 failed; it changes the status to **MARKED**, M . Meanwhile, T_3 finds the respective edge. (e) One possible linearization of this concurrent execution.

of repeated traversals by a thread. So, a **VisitedArray** slot maintains *cnt* value (see Line 116).

However, an **ENode** in **TRANSIT** state cannot be removed by any other concurrent thread other than the thread that added it, only if it creates a cycle. The threads which are performing cycle detection can see all the **ENodes** in **TRANSIT** or **ADDED** state. Further, a concurrent **ACYCONE** operation will ignore all the **ENodes** with **TRANSIT** state. This ensures that when an **ENode** is in the **ADDED** state, an **ACYADDE** operation will return **true** along with a string **EDGE ADDED**.

However, it is to be noted that with this algorithm, it is possible that an edge may not get added to the graph even though it does not create a cycle. This can happen in the following scenario; two threads T_1 and T_2 are adding edges lying in the path of a single cycle. In this case, both the threads detect that the newly added **ENode** (in **TRANSIT** state) has led to the formation of a cycle and both may delete their respective edges. However, in a sequential execution, only one of the edges would be removed. But, this implementation is correct w.r.t our sequential specification (thereby preserving our correctness criteria, linearizability) as the graph at the end of each operation remains acyclic. The proof of the acyclicity is given in the technical report [20].

Although the wait-free **SCR** algorithm does not add an edge at times even when it does not create a cycle, it can be seen its working is non-trivial. A trivial algorithm can always return **false** for **ADDEGE** while not violating the specification and hence satisfying linearizability. **SCR** algorithm is much stronger and allows insertion of edges even in the presence of concurrent updates, as explained in the working.

4.4 Obstruction-Free Double Collect Reachable Algorithm

In this subsection, we present an obstruction-free reachability, **DCR** algorithm, which is designed based on the atomic snapshot algorithm by Afek et al. [1] and

reachable algorithm by Chatterjee et al. [6]. There is no non-determinism in the DCR algorithm. It never fails to add an edge if the edge does not create a cycle. However, unlike wait-free SCR, DCR is obstruction-free. It returns only in the absence of any other concurrent updates.

The DCR (k, l) algorithm, in Lines 167 to 175, performs a SCAN starting from k . It checks whether l is reachable from k . This reachable information is returned to the ACYADDE operation and then ACYADDE decides whether to add $e(k, l)$ (is in the TRANSIT state) to the edge-list of k .

The SCAN method, in Lines 176 to 191, first creates two BFS-trees, $otree$ and $ntree$ to hold the VNodes in two consecutive BFS traversal. It performs repeated BFS-tree collection by invoking BFSTREECOLLECT until two consecutive collects are the same. The BFSTREECOLLECT procedure, in Lines 138 to 166, performs a non-recursive BFS traversal starting from the vertex k . In the process of BFS traversal, it explores all the reachable and unmarked VNodes through adjacent ENodes which are in the TRANSIT or ADDED or unmarked state. However, it keeps adding all these VNodes in the $bTree$ (see Line 142, 152, 158). If it reaches l , then it terminates by returning $bTree$ and a reachable status **true** (Line 153) to the SCAN method. If it is unable to reach l from k after exploring all reachable VNodes, then it terminates by returning $bTree$ and a reachable status **false** (Line 165) to the SCAN method.

```

167: Operation DCR ( $k, l$ )
168:   tid  $\leftarrow$  this_thread.get_id();
169:   status  $\leftarrow$  SCAN ( $k, l, tid$ );
170:   if (status = true) then
171:     return true;
172:   else
173:     return NULL;
174:   end if
175: end Operation
-----
176: procedure SCAN ( $u, v, tid$ )
177:   list < BFSNode > otree, ntree ;
178:    $\langle otree, of \rangle \leftarrow$  BFSTREECOLLECT ( $u, v,$ 
179:    $tid$ );
180:   while (true) do
181:      $\langle ntree, nf \rangle \leftarrow$  BFSTREECOLLECT ( $u, v,$ 
182:      $tid$ );
183:     if (of = true  $\wedge$  nf = true  $\wedge$  COM-
184:     PAREPATH (otree, ntree)) then
185:       return nf;
186:     else
187:       if (of = false  $\wedge$  nf = false  $\wedge$  COM-
188:       PARETREE (otree, nt)) then
189:         return nf;
190:       end if
191:     end if
192:     of  $\leftarrow$  nf;
193:     otree  $\leftarrow$  ntree;
194:   end while
195: end procedure
-----
192: procedure COMPARETREE ( $otree, ntree$ )
193:   if (otree = NULL  $\vee$  ntree = NULL) then
194:     return false ;
195:   end if
196:   BFSNode oit  $\leftarrow$  otree.Head, nit  $\leftarrow$  ntree.Head;
197:   do
198:     if (oit.n  $\neq$  nit.n  $\vee$  oit.leccount  $\neq$ 
199:     nit.leccount  $\vee$  oit.p  $\neq$  nit.p) then return
200:     false;
201:   end if
202:   oit  $\leftarrow$  oit.next; nit  $\leftarrow$  nit.next;
203:   while (oit  $\neq$  ot.Tail  $\wedge$  nit  $\neq$  nt.Tail );
204:   if (oit.n  $\neq$  nit.n  $\vee$  oit.leccount  $\neq$  nit.leccount
205:    $\vee$  oit.p  $\neq$  nit.p) then return false;
206:   else return true ;
207:   end if
208: end procedure
-----
206: procedure COMPAREPATH ( $otree, ntree$ )
207:   if (otree = NULL  $\vee$  ntree = NULL) then
208:     return false ;
209:   end if
210:   BFSNode oit  $\leftarrow$  otree.Tail, nit  $\leftarrow$  ntree.Tail;
211:   do
212:     if (oit.n  $\neq$  nit.n  $\vee$  oit.leccount  $\neq$ 
213:     nit.leccount  $\vee$  oit.p  $\neq$  nit.p) then return
214:     false;
215:   end if
216:   oit  $\leftarrow$  oit.p; nit  $\leftarrow$  nit.p;
217:   while (oit  $\neq$  ot.Head  $\wedge$  nit  $\neq$  nt.Head );
218:   if (oit.n  $\neq$  nit.n  $\vee$  oit.leccount  $\neq$  nit.leccount
219:    $\vee$  oit.p  $\neq$  nit.p) then return false;
220:   else return true;
221:   end if
222: end procedure

```

Fig. 6. Pseudo-codes of DCR, SCAN, COMPARETREE and COMPAREPATH.

If two consecutive `BFSTREECOLLECT` method return the same boolean status value `true`, then we invoke `COMPAREPATH` to compare if the two `BFS-trees` are same. If both the trees are same, then the `SCAN` method returns `true` to `DCR` operation, which means that l is reachable from k . Then `DCR` returns `true` to the `ACYADDE` operation and subsequently `ACYADDE` deletes $e(k, l)$ by setting `enext` pointer from the `TRANSIT` state to the `MARKED` state and returns `false` (this is because $e(k, l)$ created a cycle). However, if two consecutive `BFSTREECOLLECT` methods return the same status value `false`, then we invoke `COMPARETREE` to compare if the two `BFS-trees` are same. If they are, the `SCAN` method returns `false` to the `DCR` operation which implies that l is not reachable from k . Then `DCR` returns `false` to the `ACYADDE` operation and then `ACYADDE` adds $e(l)$ by setting the `enext` pointer from the `TRANSIT` state to `ADDED` state and then it returns `true`, which confirms the acyclic property after `ACYADDE`. If two consecutive `BFSTREECOLLECT` methods return the same boolean status value `true` or `false` but do not match in the `COMPAREPATH` or `COMPARETREE`, then we discard the older `BFS-tree` and restart the `BFSTREECOLLECT`.

The `COMPAREPATH` method, in Lines 206 to 219, compares two `BFS-tree` based on the path along with the `lcount` values. It starts from the last `BFSNode` and follows the parent pointer p until it reaches to the starting `BFSNode` or any mismatch that occurred at a `BFSNode`. It returns `false` for any mismatch occurred, otherwise returns `true`. Similarly, the `COMPARETREE` method, in Lines 192 to 205, compares two `BFS-tree` based on all explored `VNodes` in the process of BFS traversal and along with the `lcount` values. It starts from the starting `BFSNode` and follows with the next pointer `next` until it reaches the last `BFSNode` or any mismatch that occurred at a `BFSNode`. It returns `false` for any mismatch occurred and otherwise returns `true`.

To capture the modifications along the path of BFS-traversal, we have an atomic counter `ecount` associated with each vertex. During any edge update operation, before $e(k, l)$ gets physically deleted, the counter `ecount` of the source vertex k is certainly incremented at Line 78 or 103 either by the operation that logically deleted the $e(k, l)$ or any edge helping operations. To verify the double collect, we compare the `BFS-tree` along with the counter.

It is to be noted that even though the `DCR` algorithm is better than `SCR` as the specification of `ACYADDE` operation does not exhibit any non-determinism, it does not exploit as much concurrency as the `SCR` algorithm. As explained, the `SCR` algorithm is wait-free without using helping descriptors, whereas `DCR` is obstruction-free. In Sect. 6, we compared the performance of both these algorithms and as expected observed that `SCR` performs better.

5 Correctness and Progress Guarantee

In this section, we prove the correctness of our concurrent acyclic graph data-structure based on *LP* [14] events inside the execution interval of each of the operations.

Theorem 1. The non-blocking concurrent acyclic graph operations are linearizable.

Theorem 2. For the presented concurrent acyclic graph algorithm, (1). The operations ACYCONV, ACYCONE and SCR are wait-free, if the vertex keys are finite, (2). The operation DCR is obstruction-free and, (3). The operations ACYADDV, ACYREMV, ACYCONV, ACYADDE, ACYREME, and ACYCONE are lock-free.

The proof of Theorems 1 and 2 can be referred to from the technical report [20].

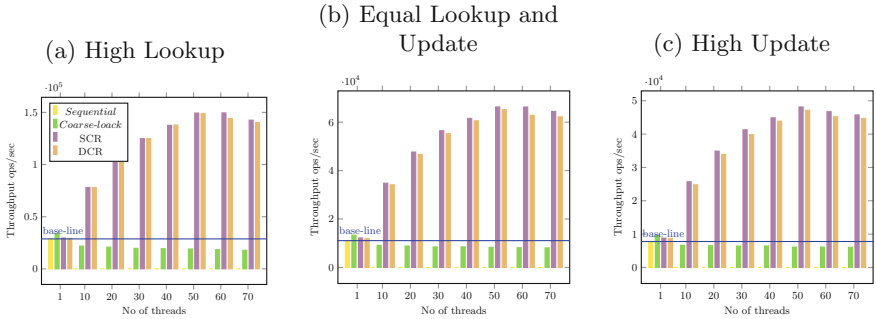


Fig. 7. Acyclic graph data-structure.

6 Experimental Evaluation

We performed our tests on 56 cores machine with Intel Xeon (R) CPU E5-2630 v4 running at 2.20 GHz frequency. Each core supports 2 logical threads. Every core's L1 cache has 64k, L2 has 256k cache memory private to that core; L3 cache (25 MB) is shared across all cores of a processor. The tests were performed in a controlled environment, where we were the sole users of the system. The implementation^a has been done in C++ (without any garbage collection) and threading is achieved by using Posix threads. All the programs were optimized at -O3 level.

We start our experiments by creating an initial directed graph with 1000 vertices and nearly $\binom{1000}{2}/4 \approx 125000$ edges added randomly. Then we create a fixed number of threads with each thread randomly performing a set of operations chosen by a particular workload distribution. We evaluate the number of operations finished their execution in unit time and then calculate the throughput. We run each experiment for 20 seconds and the final data point values are collected after taking an average of 7 iterations. We present the results for the following workload distributions for acyclic directed graph over the ordered set of operations {ACYADDV, ACYREMV, ACYCONV, ACYADDE, ACYREME, ACYCONE}

^a The source code is available on <https://github.com/PDCRL/ConcurrentGraphDS>.

as: (1) *High Lookup*: (2.5%, 2.5%, 45%, 2.5%, 2.5%, 45%), see the Fig. 7a. (2) *Equal Lookup and Update*: (12.5%, 12.5%, 25%, 12.5%, 12.5%, 25%), see the Fig. 7b. (3) *High Update*: (22.5%, 22.5%, 5%, 22.5%, 22.5%, 5%), Fig. 7c.

From Fig. 7, we notice that both SCR and DCR algorithms perform well with the number of threads in comparison with sequential and coarse-lock based version. The wait-free single collect reachable algorithm performs better than the obstruction-free double collect reachable algorithm. However, we notice that the performance of the coarse lock-based algorithm decreases with the number of threads. Moreover also, it performs worse than even the sequential implementation. On average, both the non-blocking algorithms are able to achieve nearly $7\times$ times higher throughput over the sequential implementation.

7 Conclusion

In this paper, we presented two efficient non-blocking concurrent algorithms for maintaining acyclicity in a directed graph where vertices & edges are dynamically inserted and/or deleted. The first algorithm is based on a wait-free reachability query, SCR, and the second one is based on partial snapshot-based obstruction-free reachability query, DCR. Both these algorithms maintain the acyclic property of the graph throughout the concurrent execution. We prove that the acyclic graph data-structure operations are linearizable. We also present a proof to show that the graph remains acyclic at all times in the concurrent setting. We evaluated both the algorithms in C++ implementation and tested through a number of micro-benchmarks. Our experimental results show that our proposed algorithms obtain an average of $7x$ improvement over the sequential implementation and the coarse lock based ones.

In spite of the performance of the SCR, it suffers from the non-determinism during concurrent addition of edges. It can be seen that DCR gets rid of the non-determinism and makes sure that an edge surely gets added if it does not create a cycle. In the future, we plan to measure the number of false positives incurred by the SCR algorithm on varying workloads and suggest ways to reduce them.

Acknowledgements. We would like to thank the anonymous reviewers and our shepherd, C. Aiswarya for their useful suggestions and comments. Following their suggestions, we made several improvements to the manuscript. This research was funded by MediaLab Asia for funding Graduate Scholarship.

References

1. Afek, Y., Attiya, H., Dolev, D., Gafni, E., Merritt, M., Shavit, N.: Atomic snapshots of shared memory. *J. ACM* **40**(4), 873–890 (1993)
2. Barnes, G.: A method for implementing lock-free shared-data structures. In: *Proceedings of the Fifth Annual ACM Symposium on Parallel Algorithms and Architectures, SPAA 1993*, pp. 261–270. ACM, New York (1993)

3. Brito, J., O'Sullivan, A.: Bitcoin: A Primer for Policymakers. Mercatus Center at George Mason University (2013)
4. Buterin, V.: Ethereum: a next generation smart contract and decentralized application platform (2013). <https://github.com/ethereum/wiki/wiki/white-paper>
5. Chatterjee, B., Nguyen, N., Tsigas, P.: Efficient lock-free binary search trees. In: Proceedings of the 2014 ACM Symposium on Principles of Distributed Computing, PODC 2014, pp. 322–331. ACM, New York (2014)
6. Chatterjee, B., Peri, S., Sa, M., Singhal, N.: A simple and practical concurrent non-blocking unbounded graph with linearizable reachability queries. In: ICDCN 2019, pp. 168–177 (2019)
7. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms. MIT Press, Cambridge (2009)
8. Dang, N.N., Tsigas, P.: Progress guarantees when composing lock-free objects. In: Jeannot, E., Namyst, R., Roman, J. (eds.) Euro-Par 2011. LNCS, vol. 6853, pp. 148–159. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-23397-5_15
9. Demetrescu, C., Finocchi, I., Italiano, G.F.: Dynamic graphs. In: Handbook of Data Structures and Applications, Chapman and Hall/CRC (2004)
10. Fomitchev, M., Ruppert, E.: Lock-free linked lists and skip lists. In: Proceedings of the Twenty-Third Annual ACM Symposium on Principles of Distributed Computing, PODC 2004, St. John's, Newfoundland, Canada, 25–28 July 2004, pp. 50–59 (2004)
11. Harris, T.L.: A pragmatic implementation of non-blocking linked-lists. In: Welch, J. (ed.) DISC 2001. LNCS, vol. 2180, pp. 300–314. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-45414-4_21
12. Hendler, D., Shavit, N., Yerushalmi, L.: A scalable lock-free stack algorithm. In: Proceedings of the Sixteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures, SPAA 2004, pp. 206–215. ACM, New York (2004)
13. Herlihy, M., Shavit, N.: On the nature of progress. In: Fernández Anta, A., Lipari, G., Roy, M. (eds.) OPODIS 2011. LNCS, vol. 7109, pp. 313–328. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-25873-2_22
14. Herlihy, M.P., Wing, J.M.: Linearizability: a correctness condition for concurrent objects. ACM Trans. Program. Lang. Syst. **12**(3), 463–492 (1990)
15. Kallimanis, N.D., Kanellou, E.: Wait-free concurrent graph objects with dynamic traversals. In: OPODIS 2015, pp. 27:1–27:17 (2015)
16. Ladan-Mozes, E., Shavit, N.: An optimistic approach to lock-free FIFO queues. Distrib. Comput. **20**(5), 323–341 (2008)
17. Michael, M.M.: High performance dynamic lock-free hash tables and list-based sets. In: SPAA, pp. 73–82 (2002)
18. Nakamoto, S.: Bitcoin: a peer-to-peer electronic cash system (2008). <http://bitcoin.org/bitcoin.pdf>
19. Natarajan, A., Mittal, N.: Fast concurrent lock-free binary search trees. In: ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2014, Orlando, FL, USA, 15–19 February 2014, pp. 317–328 (2014)
20. Peri, S., Sa, M., Singhal, N.: A pragmatic non-blocking concurrent directed acyclic graph. CoRR, abs/1611.03947 (2016)
21. Serguei Popov. The tangle (2018). <https://iota.org/iotawhitepaper.pdf>
22. Sinha, A., Malik, S.: Runtime checking of serializability in software transactional memory. In: IPDPS 2010, pp. 1–12 (2010)

23. Valois, J.D.: Lock-free linked lists using compare-and-swap. In: Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing, Ottawa, Ontario, Canada, 20–23 August 1995, pp. 214–222 (1995)
24. Weikum, G., Vossen, G.: Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery. Morgan Kaufmann, Burlington (2002)