View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Research Archive of Indian Institute of Technology Hyderabad

Maintaining Acyclicity of Concurrent Graphs*

Sathya Peri, Muktikanta Sa, Nandini Singhal
Department of Computer Science & Engineering
Indian Institute of Technology Hyderabad
{sathya_p, cs15resch11012, cs15mtech01004}@iith.ac.in

arXiv:1611.03947v1 [cs.DC] 12 Nov 2016

Abstract

In this paper, we consider the problem of preserving acyclicity in a directed graph (for shared memory architecture) that is concurrently being updated by threads adding/deleting vertices and edges. To the best of our knowledge, no previous paper has presented a concurrent graph data structure. We implement the concurrent directed graph data-structure as a concurrent adjacency list representation. We extend the lazy list implementation of concurrent linked lists for maintaining concurrent adjacency lists. There exists a number of graph applications which require the acyclic invariant in a directed graph. One such example is Serialization Graph Testing Algorithm used in databases and transactional memory. We present two concurrent algorithms for maintaining acyclicity in a concurrent graph: (i) Based on obstruction-free snapshots (ii) Using wait-free reachability. We compare the performance of these algorithms against the coarse-grained locking strategy, commonly used technique for allowing concurrent updates. We present the speedup obtained by these algorithms over sequential execution. As a future direction, we plan to extend this data structure for other progress conditions.

Keywords: concurrent graphs; acyclicity; lazy list; directed graph; obstruction free; wait-free

1 Introduction

As we know, graphs are present everywhere and many common real world objects can be modeled as graphs. The graphs represent pairwise relationships between objects along with their properties. Graphs are being used in various fields like: genomics, networks, coding theory, scheduling, computational devices, networks, organization of similar and dissimilar objects, etc. The current trends of research on graphs are on: social networks, semantic networks, ontology, protein structure [12], etc. Generally, these graphs are very *large* and *dynamic* in nature. Dynamic graphs are those which are subjected to a sequence of updates like insertion or deletion of vertices or edges [9]. Online social networks (facebook, linkedin, google+, twitter, quora, etc.), are dynamic by nature. The problems being addressed in the context of dynamic graphs are: finding cycles, graph coloring, minimum spanning tree, shortest path between a pair of vertices, strongly connected components, etc. There are many efficient well-known algorithms for solving these problems in the sequential world. This problem has been well explored even in the area of distributed systems. Distributed graph analytics are used to perform computations on extremely large graphs with billions of vertices and hundreds of billions of edges in a distributed manner. However, in the concurrent setting (where multiple threads access the shared memory data structure concurrently allowing synchronization via locks), to the best of our knowledge, there

*Work in Progress

is no efficient concurrent data structure that captures this inherent nature of concurrent updates to a graph.

We have been specifically motivated by the problem of Serialization Graph Testing (SGT) [16, 28, 30] in the context of Software Transactional Memory (STM) systems. A STM system handles the concurrency control over set of software transactions running concurrently. A transaction is a block of code accessing shared memory variables, which should execute atomically. SGT is a scheduling algorithm which maintains a conflict graph over all transactions. A conflict graph characterizes every transaction as a vertex and all conflicts as directed edges that respect the ordering of the conflicting transactions [30]. The conflict graph gets modified dynamically by addition or deletion of vertices and edges arriving at the scheduler.

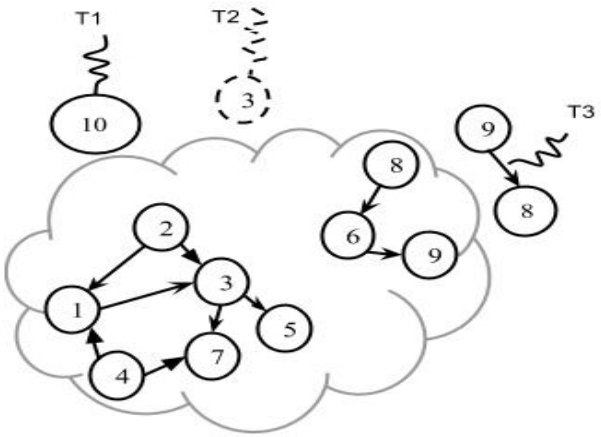

Figure 1: An example of a directed acyclic graph in the shared memory which is being accessed by multiple threads. Thread 1 is trying to add a vertex 10 to the graph. Thread 2 is concurrently invoking a remove vertex 3 call. Thread 3 is also concurrently performing an addition of directed edge from vertex 9 to vertex 8 and which will later create a cycle.

The *correctness - opacity* [13] property is maintained by ensuring that the conflict graph remains acyclic. When a set of transactions arrive concurrently at the scheduler, then the dynamic conflict graph can be tested for cycles by its global synchronization (coarse locks). This implementation can be speeded up by improving the concurrency among threads by providing finer granularities of synchronization. However, to the best of our knowledge, no previous work has explored this problem in the concurrent world.

A conceivable solution to the sequential cycle detection in directed graphs is to use depth first search and check for any back edges. This gives linear time complexity for static graphs, which performs well even for large graphs. This gets complicated if the graph is huge and multiple threads are concurrently accessing the graph and performing some operations as depicted in Figure 1. Hence, there is a need to define a correct concurrent graph data-structure to support dynamism with proper synchronization. Therefore, in this paper, we show how to construct a fully dynamic concurrent graph data structure, which allows threads to concurrently add/delete vertices/edges. Moreover, we also present two algorithms for preserving acyclicity of this concurrent graph.

The main contributions of this paper are as follows:

- A concurrent directed graph data structure represented by adjacency list that has been implemented as a concurrent set based on linked list [17].
- Two algorithms: obstruction-free snapshots & wait-free reachability for detecting cycle in the fully dynamic concurrent graph.

- Experimental analysis of the presented concurrent data structure under varying workload distributions which demonstrate the concurrency obtained against the coarse locking strategy.
- A proof sketch for proving the linearizability of all the presented methods by using *linearization points* and using invariants. The proof-sketches illustrate the properties of the concurrent graph and its acyclicity. We also give a proof-sketch of the *progress* conditions of our methods.

The organization of the paper is as follows: The section 2 describes the system model and some related work. In the section 3, we define the problem being addressed along with the underlying assumptions. The section 4 describes the construction of a fully dynamic concurrent list based adjacency list data structure along with the working of each of its methods. Section 5 presents the solution approaches for preserving acyclicity of this dynamically changing graph. Here, we also provide a proof sketch of the progress guarantees these methods provide. Results and experimental analysis are given in the section 6 and finally we conclude and present the future direction in the section 7. The Appendix at the end of the paper describes the proof sketch of the correctness of the presented algorithms and also furnishes the pseudo-codes of all the described algorithms.

2 Preliminaries

2.1 System Model

In this paper, we assume that our system consists of n processors, accessed by p threads that run in a completely asynchronous manner and communicate using shared memory. Consequently, we make no assumption about the relative speeds of the threads. We also assume that none of these processors and threads fails.

2.2 Linearizability

To prove a concurrent data structure to be correct, *linearizability* is a popular correctness criterion in the concurrent world. The history is defined as a collection of set of invocation and response events. Each invocation of a method call has a subsequent response. Herlihy and Wing [21] define a history to be *linearizable* if,

1. The invocation and response events can be reordered to get a valid sequential history.
2. The generated history satisfies the object's sequential specification.
3. If a response event precedes an invocation event in the original history, then this should be preserved in the sequential reordering.

A concurrent object is linearizable iff each of its histories is linearizable.

2.3 Progress Conditions

In this subsection, we briefly describe various progress conditions for the methods of a concurrent object as defined in the literature [20]. Some of the progress guarantees which are used in this paper are mentioned as follows:

(1) ***Blocking***: In this, an arbitrary and unexpected delay by any thread (say, one holding a lock) can prevent other threads from making progress.
(2) ***Deadlock-Free***: This is a weak blocking condition which ensures that some thread trying to acquire the lock, eventually succeeds.

(3) ***Non-blocking***: This condition ensures that threads competing for a shared resource do not have their execution indefinitely postponed by mutual exclusion.
(4) ***Wait-free***: This is a non-blocking condition in which a method is wait-free if every thread that calls that method eventually returns within a bounded number of its own time steps.
(5) ***Obstruction-free***: A method is obstruction-free if every thread that calls that method returns if that thread executes in isolation (uninterrupted) for long enough.

Some research suggests that obstruction-free is often sufficient condition to many practical systems, and is much easier to achieve [18]. In this paper, we describe an *Obstruction-free* atomic snapshot based and *Wait-free* cycle detection on a concurrent directed graph.

2.4 Related Work

There has been a lot of work on incremental cycle detection [15,27] in the sequential environment. A nice classical result of graph theory is that a directed graph is cycle-free if and only if it has a topological order [8], or rather if the strong components of a directed graph can be ordered topologically. For static directed graphs, there are two $O(m)$-time algorithms to find a cycle or a topological order and depth-first search [22, 24]:the reverse postorder defined by such a search is a topological order if the graph is acyclic. Depth-first search extends to find the strong components in a directed graph and a topological order of them in $O(m)$ [24].

The problem of cycle detection has also been well explored in distributed systems. The problem was proposed by Chandy, et al. [7] and it has also been explored by [2, 10, 23, 25, 26]. They use local *wait-for-graphs*(WFG) [29] to detect cycle in the local copy and then probe to confirm the existence of cycle in the global WFG.

The snapshot problem was discussed by Guerraoui and Ruppert [14]. Snapshots will not be consistent if we read one value at a time [11]. The problem lies in the fact that the first value read might have become outdated by the time later values are read. Therefore there is a need of an atomic implementation of snapshot [1]. Snapshots have been well explored even in distributed systems [4, 5].

3 Problem Definition

The problem addressed in this paper is defined as here:

1. A concurrent directed graph $G = (V, E)$ is dynamically being modified by a fixed set of concurrent threads. In this setting, threads may perform insertion / deletion of vertices or edges to the graph.

2. We also maintain the invariant that the concurrent graph G updated by concurrent threads should be acyclic. This means that the graph should preserve acyclicity at the end of every operation in the generated equivalent sequential history.

An important assumption here is that the key size of vertices is finite. Also, all the vertices are assigned unique keys. Since the motivation for maintaining such a concurrent graph is Serialization Graph Testing Algorithm, we impose a constraint of not allowing same vertex id's to come again because transactions always come with increasing id's. This means that if a vertex id has been removed then it will not be added again to the concurrent graph G. However, this constraint can be lifted after making minor modifications (like using timestamps for the cycle detection method using snapshots) to the algorithm.
In this section, we describe the interface of the fully dynamic concurrent graph data structure. We represent a graph using adjacency list representation, which is a list of linked lists as depicted in the Figure 2 and 3. The underlying concurrent linked list implementation is an adaptation

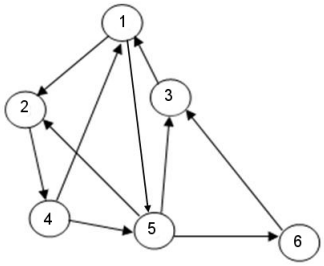

Figure 2: An example of a directed graph

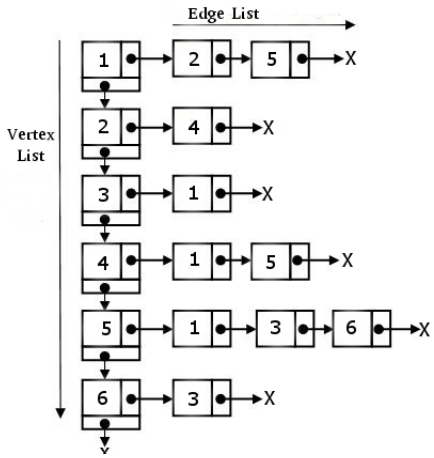

Figure 3: Adjacency List corresponding to the graph depicted in Figure 2

of the concurrent set. Each vertex list and edge list corresponding to each vertex is stored in sorted manner, similar to the set implementation.
Our concurrent directed adjacency list exports the following methods:

1. The *AddVertex(u)* method adds a vertex u to the graph, returning *true* if, and only if vertex u was not already there else returns $false$.

2. The *RemoveVertex(u)* method deletes vertex u from the adjacency list, returning *true* if, and only if u was already there else it returns $false$.

3. The *AddEdge(u, v)* method adds a directed edge (u, v) to the concurrent graph, returning *true* if, and only if (u, v) was not already present in the graph else returns $false$. It then invokes the *CycleDetect* method to verify if the inserted edge caused a cycle. If a cycle has been detected then the edge is deleted by invoking *RemoveEdge(u,v)*.

4. The *RemoveEdge(u,v)* method deletes the directed edge (u, v) from the concurrent adjacency list, returning *true* if, and only if (u, v) was already there else it returns $false$.

5. The *ContainsEdge(u,v)* returns *true* if, and only if the lazy list contains the edge (u, v) else returns $false$.

6. The *ContainsVertex(u)* returns *true* if, and only if the lazy list contains the vertex u else returns $false$.

A typical application uses significantly more *ContainsEdge(u,v)* and *ContainsVertex(u)* calls than the update calls.

4 Construction of Concurrent List based Directed Graphs

Till now, to the best of our knowledge, no concurrent adjacency list data structure has been proposed. Hence, when multiple threads simultaneously update the graph data structure, it is done by using coarse locks on the method calls. The problem with this is that when multiple threads try to access the concurrent data structure at the same time, then the data structure becomes a sequential hold-up, forcing threads to wait in line for access to the lock. This graph data structure can be used as a fundamental building unit for different granularities of synchronization. In this paper, we consider lazy synchronization of concurrent set implemented using linked lists [17] to implement dynamically changing adjacency lists data structure[1]. Instead of maintaining a coarse lock on the adjacency list as a whole, we maintain locks for individual vertex and edge nodes in the adjacency list to increase the concurrency. Lazy synchronization further increases concurrency by allowing traversals to occur without acquiring locks. Once the correct nodes have been found and locks have been acquired, then the thread validates if the locked nodes are indeed correct. If before acquiring locks, some other thread has updated the data structure and wrong nodes were locked, then the locks are released and traversals start over.

4.1 Working of the Add methods

The $AddVertex(u)$ method is blocking, with its implementation being no different from the add method in the concurrent lazy linked list. When a thread wants to add a vertex to the concurrent graph, then it traverses the vertex list without acquiring any locks until it finds a vertex with its key greater than u, say *curr*. It then acquires lock on the vertex preceding *curr* and *curr* itself and validates to check if *curr* is reachable from its predecessor, say *pred*, and both the nodes have not been deleted (marked). The algorithm maintains an invariant that all the unmarked vertex and edge nodes are reachable. If the validation succeeds, then the thread adds the vertex u between *pred* and *curr* in the vertex list and returns true after unlocking the vertices. If it fails, then the thread starts the traversal over after unlocking the locked vertices.

The $AddEdge(u, v)$ method has now been extended to first check the vertex list for $ContainsVertex(u)$ and $Contains\text{-}Vertex(v)$. Once the vertices u and v are validated to be reachable and unmarked in the vertex list, the thread then traverses the adjacency list of vertex u to add a edge node for the vertex v without acquiring locks. Once a edge node with key greater than v is encountered, say *curr*, locks are obtained on the predecessor of the *curr* edge node and *curr* node itself. After this, validation is performed to check if the respective edge nodes are unmarked and reachable. If the validation is successful, then the new edge node is added in between *pred* and *curr* in the edge list of u. If this method returns *true*, then the graph acyclic property maybe violated. To identify this, the method invokes *CycleDetect*. In section 5, we provide two solution approaches for cycle detection in the concurrent graph environment. At any instant, when a new directed edge is added to the graph, the concurrent graph is checked for maintaining acyclicity. If it does not preserve, then the edge is removed.

4.2 Working of the Remove methods

The *RemoveEdge(u,v)* method proceeds similar to the $Add\text{-}Edge(u, v)$ by traversing the vertex list in the adjacency list, without locks and then verifying that the vertices u and v are indeed present in the graph. The thread then traverses the edge list of u without acquiring locks and then validating them. Once the edge node to be deleted and its predecessor have been locked, then logical removal occurs. Logical removal involves setting the marked field of the node to true. Physical deletion of edge nodes can take place along with logical removal. This is because

[1]The complete pseudo code of the algorithms are given in the Appendix in *Listings 1 - 10*

the edges in the concurrent graph are directed and a edge needs to be removed only from one list. Physical deletion involves changing the pointer of the predecessor of the marked node to its successor so that the deleted node is no longer reachable in the edge list.

The $RemoveVertex(u)$ method proceeds similar to the *AddVertex(u)*. Once the vertex to be deleted and its predecessor have been locked, then logical removal occurs. This ensures that if any edge is being added or removed concurrently corresponding to that vertex, then it will fail in the validation process after checking the marked field. Physical deletion of a vertex node is more complicated as all edges corresponding to that vertex also need to be deleted from the concurrent graph. This is done by performing a traversal to check if the deleted vertex is contained in the adjacency list of any other vertex node. If the vertex is found, then locks are obtained on the *pred* and *curr* edge nodes respectively and the deleted edge node is marked (Logical removal). Consequently, the edge nodes are also physically deleted while locks have been acquired.

4.3 Working of the Contains methods

Methods $ContainsVertex(u)$ and $ContainsEdge(u, v)$ are wait-free and traverse the adjacency list without acquiring locks. These methods return *true* if the vertex/edge node it was searching for is present and unmarked in the adjacency list and $false$ otherwise.

It can be seen that all the update calls are blocking (use locks) and they are deadlock-free. All the methods described above are linearizable. The proof sketch is described in the Appendix Section.

5 Maintaining Graph Acyclicity

In this section, we consider the problem of maintaining an invariant of acyclicity in this concurrent dynamic graph data structure. For a concurrent graph to be acyclic, it means that the graph should maintain the acyclic property at the end of each operation in the equivalent sequential history constructed after the linearization of the concurrent history. For this, we present two algorithms for concurrent cycle detection providing different progress guarantees. It is easy to see that a cycle can be created only on addition of edge to the graph. After the edge has been added to the graph (in the shared memory), we then verify if the resulting graph is acyclic. If it is, we leave the edge. Otherwise, we delete the edge from the shared memory. The subsections here present approaches for detecting cycle in this concurrent setting of graph.

5.1 Obstruction-Free Snapshot

The atomic snapshot-object based problem was first introduced by Anderson [3] and Afek, et al. [1] in 1990, and since then substantial research work has been done in the fields of both shared and distributed memory environments. In a concurrent setting, when multiple threads are acting upon the shared memory graph data structure consisting of m components, shared by upto n concurrent threads, a thread can invoke the *update* method for adding or deleting vertices/edges. The *CycleDetect* method invokes *scan* which returns an atomic snapshot of the graph data structure. In this context, the method captures the snapshot of the entire shared memory adjacency list. Here, we consider the *obstruction-freedom* snapshot, which ensures termination only for threads that eventually execute uninterrupted by any other threads. This approach has been inspired from [4, 5] which identifies a stable property by taking consistent snapshots over a period of collected states of the computation.

The atomic obstruction-free method is described by the *scan* which invokes the *collect* method. The *collect* is the non-atomic act of copying the entire adjacency list values one-by-one into a local copy. Since the key size if finite, the adjacency lists are finite. Hence a single

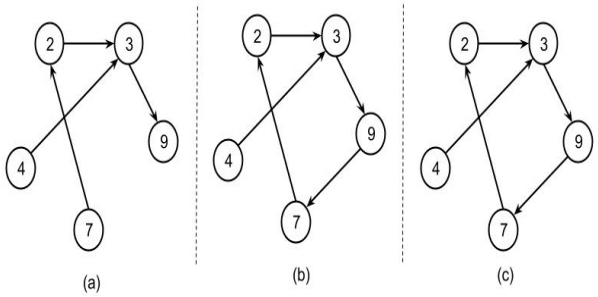

Figure 4: An execution of obstruction-free snapshot. Figure (a) is the initial snapshot collected by a thread. Figure (b) is the second snapshot by the same thread. It is seen that this snapshot and one collected before are different. Figure (c) The thread takes a snapshot for the third time and this time the snapshot is the same as the second one. Now, sequential cycle detection can be invoked on the third snapshot and a cycle is detected.

collect will terminate in finite number of steps. The *scan* method repeatedly performs two collects: *oldCopy* and *newCopy* one after the other with some time interval. If both collects read the same set of values as well as marked field, then we know that there was an interval during which no other thread updated the data structure, so the result of the collect is a snapshot of the system state immediately after the end of the first collect. Such a pair of collects is called a *clean double collect*.

A clean double collect is also correct because the data structure maintains the underlying assumption that the keys of the vertices are unique and a vertex which was once removed will never get added back with the same key. Hence this collect would exist into the shared memory at some instant. Since, a clean double collect can only be collected if the snapshot thread runs in isolation (without being interrupted by any other thread invoking updates) for the duration of two collects, this method is obstruction-free. After getting a valid collect, we invoke any sequential cycle detection algorithm (for example, one-way search algorithm [6]) on that collect. If a cycle is found, then the edge which caused the cycle is removed from the concurrent graph else the algorithm continues normally. An example of this is depicted in Figure 4 and the complete pseudo code is described in Listing 11 in the Appendix Section respectively.

5.2 Wait-Free Reachability

Definition (Reachability). Given a directed graph, $G = (V, E)$ and two vertices $u, v \in V$, a vertex v is reachable from another vertex u, iff there exists a path from the vertex u to v in G.

By using this reachability definition, we define a method for cycle detection in concurrent directed graphs. Before invoking this method, *AddEdge* has already returned true. So this is to check if there exists a path from v to u in the concurrent graph. The *reachability* method creates a local *ReachSet* of all vertices reachable from v, with a *explored* boolean field corresponding to each vertex in this set. The reachability method begins by traversing the adjacency list of v to find u in the concurrent graph, without acquiring locks. All these traversed vertices edge nodes are added to the local *ReachSet* and the vertex v is marked to be explored. Now, the method recursively visits (similar to breadth first traversal) the outgoing edges from the neighbours of v to find u. Clearly, this is done until all the vertices in all the paths from v to u in G have been *explored* or a cycle has been detected in the graph.

However, in the concurrent setting, the set of vertices in the path keep varying dynamically. Since the key size is finite and all keys are unique, the adjacency list will be traversed in finite number of steps. Also, since there can only be a finite number of vertices in the path from v to u, this method will terminate in a finite number of steps. We define *reachability* without

acquiring any locks and it terminates in a finite number of steps, so it satisfies *wait-freedom* guarantee. A side-effect to be observed here is that this method may allow *false positives*. This means that the algorithm may detect a cycle even though the graph does not contain one. This can happen in the following scenario: Two threads T_1 and T_2 are adding edges corresponding to a single cycle. In this case, both threads detect that the added edge has lead to formation of a cycle and both may invoke remove edge. However, in a sequential execution, one of the edge would have not been needed to be removed.

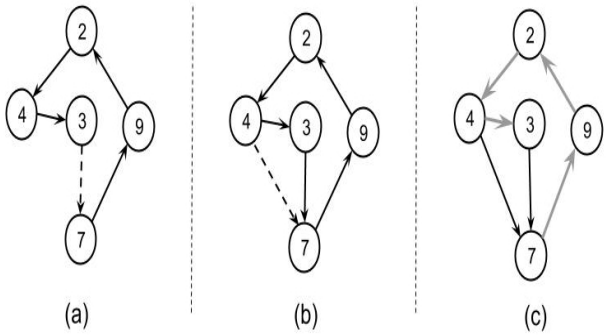

Figure 5: An execution of wait-free reachability for concurrent cycle detect. Figure (a) is the initial graph when a thread 1 is trying to concurrently add edge (3,7) to the graph. Figure (b) depicts the graph when thread 1 has finished adding edge (3,7) and is invoking cycle detect. Concurrently, thread 2 is trying to add edge (4,7). Figure (c) shows the reachability path from vertex 7 to 3 as computed by thread 1. A cycle has now been detected and hence the edge (3,7) will now be removed. Similarly, the thread adding edge (4,7) will also invoke cycle detect afterwards and remove it.

It can be easily seen that this method is a straight forward extension of the wait-free *contains* call of lazy concurrent linked list implementation. Once a cycle has been detected, then the edge which caused the cycle is removed from the concurrent graph else the algorithm proceeds normally. An execution of the algorithm is illustrated in Figure 5 and the complete pseudo code is described in Listing 12 in the Appendix Section respectively.

6 Simulation Results & Analysis

We performed our tests on 24 core Intel Xeon server running at 3.07 GHz core frequency. Each core supports 6 hardware threads, clocked at 1600 MHz.

In the experiments conducted, we start with an empty graph initially. When the program starts, it creates 150 threads and each thread randomly performs a set of operations chosen by a particular workload distribution. Here, the evaluation metric used is the time taken to complete all the operations. The update calls are implemented using lazy synchronization of concurrent linked lists whereas the cycle detection method is implemented via snapshot and reachability methods as described in the previous section. These are compared against a coarse lock implementation of the update and cycle detection methods. We do not independently invoke *RemoveEdge* method in the performance results presented here. This is invoked internally when a cycle has been detected.

Speedup is computed as the ratio of the time taken by a parallel algorithm to the sequential one. Higher the speedup, higher is the throughput. We measure speedup obtained against the sequential implementation and present the results for the following workload distributions: (a) *AddEdge-dominated*: 55% AddEdge, 25% AddVertex, 20% RemoveVertex and (b) *semi-balanced*: 30% AddEdge, 50% AddVertex, 20% RemoveVertex and the performance results are

depicted in Figure 6 and 7 respectively. Each data point is obtained after averaging for 5 iterations.

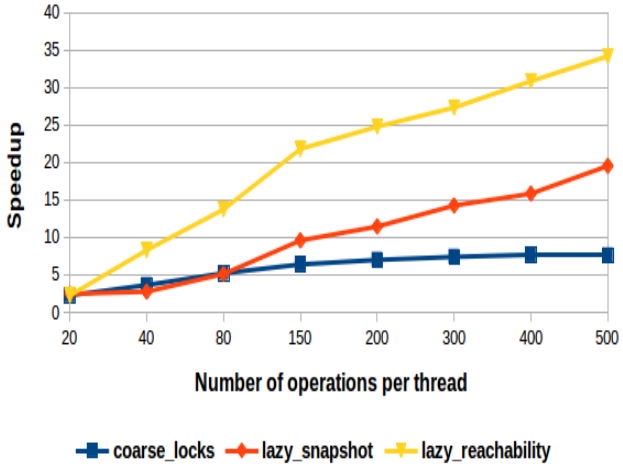

Figure 6: Speedup obtained against sequential v/s Number of operations/thread for *AddEdge-dominated* workload

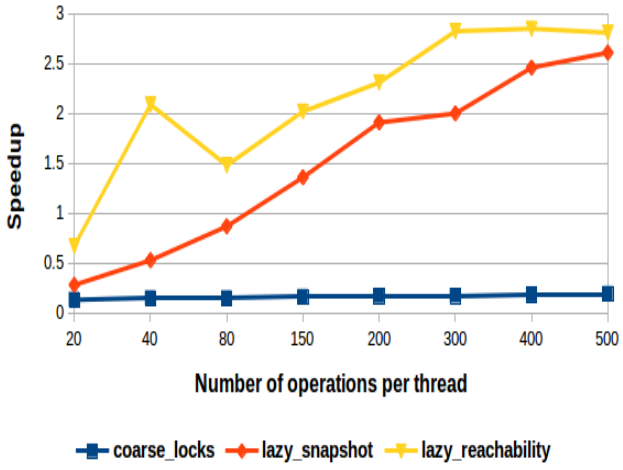

Figure 7: Speedup obtained against sequential v/s Number of operations/thread for *semi-balanced* workload

The Figure 6 shows the performance results for the increasing number of operations per thread where the larger percentage of operations comprise of addition of edges in the graph. The results depict that all the approaches perform better than the sequential approach. It is evident that lazy update calls with snapshot and reachability based cycle detection gives a significant speedup and scales well with the increasing number of operations. The Figure 7, on the other hand shows the performance results for a semi-balanced workload. As is evident, coarse locking strategy gives fixed speedup in both Figure 6 and 7. Also, reachability based method for cycle detection outperforms the snapshot based method. As can also be seen, not as much speedup is obtained in Figure 7 as in Figure 6 because of the percentage of $AddVertex$ calls being relatively more than $AddEdge$. This means that the time taken to traverse the vertex list is high in all

the algorithms (including sequential).

7 Conclusion & Future Direction

In this paper, we have shown how to construct a fully dynamic concurrent graph data structure, which allows threads to concurrently add/delete vertices/edges. To the the best of our knowledge, this is the first work to propose a concurrent data structure for an adjacency list representation of the graphs. We believe that there are many applications that can benefit from this concurrent graph structure. An important example being SGT in databases and Transactional Memory. Furthermore, on this concurrent graph data-structure, we pose the constraint that the graph should be acyclic. We ensure this by checking graph acyclicity whenever we add an edge. To detect the cycle efficiently we have proposed two algorithms, the *Obstruction-free* based atomic snapshot and *Wait-free* reachability. Among these both, the latter one performs better. We have compared the performance of both algorithms with the coarse-gained locking implementation and both of performed relatively good. For proving the correctness of our algorithm, we have used the linearizability property and illustrated the proof sketch using linearization points in the Appendix section.

In the future, we plan to develop a concurrent graph data structure satisfying wait-free progress conditions. We also plan to test the results by lifting the constraint of not allowing vertices and edges to come again with the same key, once they have been deleted. This can be easily achieved by using the notion of timestamps. The concurrency in *wait-free* reachability algorithm can be further increased by using two-way searching technique. We also plan to think of other ways of cycle detection by extending ideas of incremental cycle detection algorithms [6] to the concurrent setting. Furthermore, the obstruction-free snapshot based cycle detection can also be extended to wait-free atomic snapshot and provide stronger progress conditions.

References

[1] Y. Afek, H. Attiya, D. Dolev, E. Gafni, M. Merritt, and N. Shavit. Atomic snapshots of shared memory. *J. ACM*, 40(4):873–890, 1993.

[2] D. Ambroise, B. Rozoy, and J. Saquet. Deadlock detection in distributed systems. In *Proceedings of the ISCA 18th International Conference Computers and Their Applications, Honolulu, Hawaii, USA, March 26-28, 2003*, pages 210–213, 2003.

[3] J. H. Anderson. Composite registers. *Distributed Computing*, 6(3):141–154, 1993.

[4] R. Atreya, N. Mittal, and V. K. Garg. Detecting locally stable predicates without modifying application messages. In *Principles of Distributed Systems, 7th International Conference, OPODIS 2003 La Martinique, French West Indies, December 10-13, 2003 Revised Selected Papers*, pages 20–33, 2003.

[5] R. Atreya, N. Mittal, A. D. Kshemkalyani, V. K. Garg, and M. Singhal. Efficient detection of a locally stable predicate in a distributed system. *J. Parallel Distrib. Comput.*, 67(4):369–385, 2007.

[6] M. A. Bender, J. T. Fineman, S. Gilbert, and R. E. Tarjan. A new approach to incremental cycle detection and related problems. *ACM Trans. Algorithms*, 12(2):14, 2016.

[7] K. M. Chandy, J. Misra, and L. M. Haas. Distributed deadlock detection. *ACM Trans. Comput. Syst.*, 1(2):144–156, May 1983.

[8] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms (3. ed.)*. MIT Press, 2009.

[9] C. Demetrescu, I. Finocchi, and G. F. Italiano. Dynamic graphs. In *Handbook of Data Structures and Applications.* 2004.

[10] A. K. Elmagarmid. A survey of distributed deadlock algorithms. *SIGMOD Record*, 15(3):37–45, 1986.

[11] F. E. Fich, V. Luchangco, M. Moir, and N. Shavit. Obstruction-free algorithms can be practically wait-free. In *Distributed Computing, 19th International Conference, DISC 2005, Cracow, Poland, September 26-29, 2005, Proceedings*, pages 78–92, 2005.

[12] J. A. Gallian. A dynamic survey of graph labeling. *The Electronic Journal of Combinatorics*, 6, 2000.

[13] R. Guerraoui and M. Kapalka. On the correctness of transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2008, Salt Lake City, UT, USA, February 20-23, 2008*, pages 175–184, 2008.

[14] R. Guerraoui and E. Ruppert. Anonymous and fault-tolerant shared-memory computing. *Distributed Computing*, 20(3):165–177, 2007.

[15] B. Haeupler, T. Kavitha, R. Mathew, S. Sen, and R. E. Tarjan. Incremental cycle detection, topological ordering, and strong component maintenance. *ACM Trans. Algorithms*, 8(1):3, 2012.

[16] T. Harris, S. Marlow, S. L. P. Jones, and M. Herlihy. Composable memory transactions. *Commun. ACM*, 51(8):91–100, 2008.

[17] S. Heller, M. Herlihy, V. Luchangco, M. Moir, W. N. S. III, and N. Shavit. A lazy concurrent list-based set algorithm. *Parallel Processing Letters*, 17(4):411–424, 2007.

[18] M. Herlihy, V. Luchangco, and M. Moir. 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*, pages 522–529, 2003.

[19] M. Herlihy and N. Shavit. *The art of multiprocessor programming.* Morgan Kaufmann, 2008.

[20] M. Herlihy and N. Shavit. On the nature of progress. In *Principles of Distributed Systems - 15th International Conference, OPODIS 2011, Toulouse, France, December 13-16, 2011. Proceedings*, pages 313–328, 2011.

[21] M. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.

[22] A. B. Kahn. Topological sorting of large networks. *Commun. ACM*, 5(11):558–562, 1962.

[23] E. Knapp. Deadlock detection in distributed databases. *ACM Comput. Surv.*, 19(4):303–328, Dec. 1987.

[24] D. E. Knuth. *The Art of Computer Programming, Volume I: Fundamental Algorithms.* Addison-Wesley, 1968.

[25] S. Lee and J. L. Kim. An efficient distributed deadlock detection algorithm. In *Proceedings of the 15th International Conference on Distributed Computing Systems, Vancouver, British Columbia, Canada, May 30 - June 2, 1995*, pages 169–178, 1995.

[26] D. P. Mitchell and M. J. Merritt. A distributed algorithm for deadlock detection and resolution. In *Proceedings of the Third Annual ACM Symposium on Principles of Distributed Computing*, PODC '84, pages 282–284, New York, NY, USA, 1984. ACM.

[27] D. J. Pearce, P. H. J. Kelly, and C. Hankin. Online cycle detection and difference propagation for pointer analysis. In *3rd IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2003), 26-27 September 2003, Amsterdam, The Netherlands*, pages 3–12, 2003.

[28] H. E. Ramadan, I. Roy, M. Herlihy, and E. Witchel. Committing conflicting transactions in an STM. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2009, Raleigh, NC, USA, February 14-18, 2009*, pages 163–172, 2009.

[29] A. Silberschatz, P. B. Galvin, and G. Gagne. *Operating system concepts, 7th Edition.* Wiley, 2005.

[30] G. Weikum and G. Vossen. *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery.* Morgan Kaufmann, 2002.

Correctness

In this section, we present a proof sketch of the correctness of our proposed concurrent graph data structure and the cycle detection algorithms. Here, we prove the following properties:
(1) the concurrent graph implementation is linearizable,
(2) it is *deadlock-free*,
(3) the cycle detection algorithm proposed in section 5.1 is *obstruction-free* and in section 5.2 is *wait-free*

The following subsections provide a proof sketch of each one in detail.

Linearizability

To prove a concurrent data structure to be correct, *linearizability* is a popular correctness criterion in the concurrent world. The history is defined as a collection of set of invocation and response events. Each invocation of a method call has a subsequent response. Herlihy and Wing [21] defines a history to be linearizable if,

1. The invocation and response events can be reordered to get a valid sequential history.
2. The generated history satisfies the object's sequential specification.
3. If a response event precedes an invocation event in the original history, then this should be preserved in the sequential reordering.

A concurrent object is linearizable iff each of their histories is linearizable. Linearizability ensures that every concurrent execution can be proven by sequential execution of that object and it helps to determine the order of *linearization points* in the concurrent execution.

Linearization Point: We prove the linearizability of a concurrent history by defining a *linearization point* for each method call at some instant between invocation and response event [19]. This means that each method appears to occur instantly at its linearization point, and the behaviour is exactly same as defined by the sequential specification.

The linearization point for all the methods are given as below:

- The linearization point for the $AddEdge(u, v)$ (Listing 7) call depends on whether the method returned successfully by the thread. In each case, the linearization point is different. If the vertex u is absent from the vertex list, then linearization point is $Line8$. If the vertex v is absent from the vertex list, then point is $Line16$. And if the edge (u, v) was already present in the graph, then the linearization point for this method happens at $Line30$. Also if the edge (u, v) is absent, then the linearization point occurs when the node with the subsequent greater key has been locked i.e. $Line24$.
- The linearization point for the $RemoveEdge(u, v)$ (Listing 8) depends on whether the call was unsuccessful by a thread. It has four possibilities of linearization point first one, if the vertex u is absent in the list, then linearization occurs at $Line9$. If the vertex v is absent in the vertex list, then linearization occurs at $Line17$ third one. However, if the edge (u, v) is absent in the list, then the linearization point is $Line31$. And if the edge (u, v) is present, then the linearization point is when the node with the subsequent greater key has been locked i.e. $Line25$ or the marked field is set to true is $Line34$.
- The linearization point for the $AddVertex(u)$ (Listing 9) depends on whether the call was successful by a thread. If the vertex u is present in the list, then the linearization is $Line15$ second one, the vertex u is absent in the list is the linearization point when the node with the subsequent greater key has been locked $Line9$.

- The linearization point for the $RemoveVertex(u)$ (Listing 10) call depends on whether it was an unsuccessful called by a thread. It has two possibilities of linearization point first one, whether the vertex u is absent in the list is $Line15$ second one, the vertex u is present when the node with the subsequent greater key has been locked $Line9$ or the marked field is set to true is $Line19$.

- The linearization point for the $ContainsEdge(u, v)$ (Listing 5) method varies according to whether the vertex u is present and it's marked field set to true. If so, then linearization occurs at $Line7$. However, if the vertex v is present and it's marked field set to true is $Line13$. For a successful call, the edge (u, v) is present and the vertex v is not marked, then linearization occurs at $Line17$.

- The linearization point for the $ContainsVertex(u)$ (Listing 6) method depends on whether the call was successfully returned by the thread. The linearization point of this method occurs at return step i.e. $Line5$.

Deadlock freedom

The blocking update methods to the concurrent graph data structure proposed in section 4 are deadlock-free. This is because the underlying list data structure is implemented using a set which maintains the keys in sorted order. This means that a thread will always acquires locks either on a $ENode$ or a $VNode$ with smaller keys first. That means, when a thread is trying to acquire a lock on a node (either in the vertex list or in the edge list) with a key x, then this thread has never tried to obtain a lock on a node which has a key greater than x. From this, we can see that all the methods (AddEdge, RemoveEdge, AddVertex or RemoveVertex) can only acquire the locks on their predecessor node and then its successor. Some thread can progress to carry out it's operation and eventually succeeds. Hence, the presented algorithms are *deadlock-free*.

Obstruction-free and Wait-free cycle detection

Lemma 1: If a thread which has a *scan* method and it makes a successfully *clean double collect* in case of obstruction free atomic snapshot, then the result it returns are the values that exist in the concurrent graph data structure in some state of the execution.

Proof: The scan method repeatedly performs two collects: *oldCopy* and *newCopy* one after the other with some time interval. Consider a time interval between the last reads of oldCopy collect, and the first read of newCopy collect. If any thread were updated the graph data structure in that time interval, the oldCopy and newCopy would not match, and hence the *clean double collect* would not be true.

Lemma 2: If a thread A has *scan* method and it notices that there is a change by thread B during two consecutive double collects, then the $B's$ reading value during the last collect was written by an update method call which began only after the the first of the four collects started.

Proof: As thread A has *scan* method and it is reading $B's$ update values. In the two consecutive reads of A, B written at least once between $A's$ pair of reads. As thread B is updating the data structure, it writes it's value in it's final step, hence some $B's$ update method calls ends after the first read by thread A. Then the write step of the other happened in between the last pair of reads by thread A. The proof follows because only the thread B is writing to the data structure.

Lemma 3: The result returned by a *scan* in obstruction-free snapshot method were in the graph data structure at some state in between the invocation and response call.

Proof: If the a thread A has a *scan* method and it's call have a successful *clean double collect*, than the claim follows from Lemma 1. And if this call took the scan values from another thread $B's$ update, then by Lemma 2, the scan value found in the $B's$ data structure was acquired by the *scan* call of the thread B whose interval lies between $A's$ first and second read of $B's$ data structure. In the above case either B's *scan* call had a clean double collect, in which the result follows form Lemma 1 or there may be another thread C doing it's operation within the intervals of $B's$ *scan* call. Inductively we can apply the above argument and not more than $n-1$ nested call, where n is the maximum number of threads. At the end some nested *scan* call must have a *clean double collect*.

Lemma 4: In case of obstruction-free snapshot every *scan* method returns after at most $O(k)$ steps, where k is the key size of the graph.

Proof: As we have fixed number of threads n, for a given *scan* a thread try to get double collects, if two consecutive collects are matched then it will return clean double collect, if not it will scan all the keys present in the list. As we are using the fixed key size k, so the *scan* method returns at most $O(k)$ steps.

Lemma 5: The *wait-free* reachability cycle detection returns after at most $O(V+E)$ steps, where V, is the number of vertices and E is the number of edges in the graph.

Proof: The reachability wait-free cycle detection method called only after *AddEdge* returns true. There are at most V vertices in the graph and E edges in the graph. In the worst case each thread needs to visit all the vertices and edges. So the worst case running time be $O(V+E)$.

Lemma 6: The *readonly* methods:ContainsEdge and ContainsVertex returns after at most $O(V)$ steps.

Proof: As the *ContainsEdge* and *ContainsVertex* were wait-free. Neither of these methods were using locks nor they were updating the data structure. So, *ContainsVertex* returns after at most $O(V)$ and *ContainsEdge* returns after at most $V-1$ steps. so the running time is $O(V)$.

Pseudo Codes: Lazy Synchronization of Concurrent Graphs

Listing 1: NodeInfo Class

```
1 class NodeInfo{
2    public int key;
3    public volatile boolean marked;
4  }
```

Listing 2: EdgeNode Class

```
1 class ENode{
2    NodeInfo info;
3    ENode next;
4    Lock lock;
5    ENode(int key){//initial constructor
6       this.info.key = key;
```

```
7         this.info.marked = false;
8         this.next = null;
9         this.lock = new ReentrantLock();
10     }
11     void lock(){
12        lock.lock();
13     }
14     void unlock(){
15        lock.unlock();
16     }
17 }
```

Listing 3: VertexNode Class

```
1  class VNode{
2     NodeInfo info;
3     ENode listhead;/*holding edge node*/
4     VNode next;/*holding next vertex node*/
5     Lock lock;
6     VNode(int key){//initial constructor
7        this.info.key = key;
8        this.info.marked = false;
9        this.next = null;
10       this.listhead = null;
11       this.lock = new ReentrantLock();
12    }
13    void lock(){
14       lock.lock();
15    }
16    void unlock(){
17       lock.unlock();
18    }
19 }
```

Listing 4: Validation Class

```
1  private boolean validateVertex(VNode pred, VNode curr){
2      return  !pred.info.marked && !curr.info.marked && pred.next == curr;
3    }
4
5  private boolean validateEdge(ENode pred, ENode curr){
6      return  !pred.info.marked && !curr.info.marked && pred.next == curr;
7    }
```

Listing 5: Wait-Free Contains (Edge)

```
1   public boolean ContainsEdge(int keyu, int keyv){
2      VNode curru = head;
3      while (curru.key < keyu)
4         curru = curru.next;
5      /*searching for vertex keyu*/
6      if(curru.info.key != keyu || curru.info.marked)
7         return false;
8      VNode curr = head;
9      while (curr.key < keyv)
10        curr = curr.next;
11     /*searching for vertex keyv*/
12     if(curr.info.key != keyv || curr.info.marked)
13        return false;
14     ENode currv = curru.listhead;
15     while (currv.keyv < keyv)
```

```
16          currv = currv.next;
17      return currv.keyv == keyv && !currv.marked;
18  }
```

Listing 6: Wait-Free Contains (Vertex)

```
1   public boolean ContainsVertex(int keyu){
2     VNode curru = head;
3     while (curru.info.key < keyu)
4        curru = curru.next;
5     return curru.info.key == keyu && !curru.info.marked;
6   }
```

Listing 7: Add Edge

```
1  public boolean AddEdge(int keyu, int keyv){
2     while(true) {
3        VNode curru = head;
4        while (curru.info.key<keyu){/*traversing vertex list*/
5          curru = curru.next;
6        }
7        if(curru.info.key != keyu || curru.info.marked)
8           return false; /*vertex keyu is not present, stop adding edge*/
9        VNode currv = head;
10       /*traversing vertex list */
11       while(currv.info.key < keyv){
12          currv = currv.next;
13       }
14 /*check if vertex keyv is present in the vertex list */
15       if(currv.info.key != keyv || currv.info.marked)
16           return false;  /*v is not present, stop adding edge*/
17 /*traversal through edge list of vertex keyu*/
18      ENode predv = curru.listhead;
19      ENode currv = curru.listhead.next;
20        while (currv.info.key < keyv){
21           predv = currv;
22           currv = currv.next;
23        }
24        predv.lock();
25        try {
26           currv.lock();
27           try {
28              if(validateEdge(predv, currv)) {
29                 if (currv.info.key == keyv) {/*already present*/
30                    return false;
31                } else { /*not present*/
32                   ENode newNode = new ENode(keyv);
33                   newNode.next = currv;
34                   predv.next = newNode;
35                   return true;
36              /*Invoke the cycle detect
37               if the CycleDetect() returns true invoke the remove edge
                     RemoveEdge(keyu,keyv)
38               else addEdge succeeds.
39              */
40               }
41
42              }
43              }
44         } finally {/*always unlock*/
45            currv.unlock();
46         }
```

```
47     } finally { /*always unlock*/
48        predv.unlock();
49      }
50    } /*end of while(true)*/
51 }
```

Listing 8: Delete Edge

```
1  public boolean RemoveEdge(int keyu, int keyv){
2    while (true) {
3      VNode curru = head;
4      /*traversing vertex list */
5      while (curru.info.key < keyu){
6          curru = curru.next;
7      }
8      if(curru.info.key != keyu || curru.info.marked)
9          return false;/*vertex keyu is not present, stop deleting edge*/
10     VNode currv = head;
11      /*traversing vertex list */
12     while(currv.info.key < keyv){
13         currv = currv.next;
14     }
15     /*check for v is present the the list*/
16     if(currv.info.key != keyv || currv.info.marked)
17         return false;/*vertex keyv is not present, stop deleting edge*/
18     /*traversal through edge list*/
19     ENode predv = curru.listhead;
20     ENode currv = curru.listhead.next;
21     while (currv.info.key < keyv){
22        predv = currv;
23        currv = currv.next;
24     }
25     predv.lock();
26     try{
27        currv.lock();
28        try{
29           if(validateEdge(predv, currv)){
30            if(currv.info.key != keyv) {/*edge not present*/
31               return false;
32            }else {
33               currv.info.marked = true;/*logical removal*/
34               predv.next = currv.next;/*physical removal*/
35               return true;
36            }
37           }
38         } finally{/*always unlock*/
39            currv.unlock();
40         }
41       } finally {/*always unlock*/
42         predv.unlock();
43     }
44   }/*end of while(true)*/
45 }
```

Listing 9: Add Vertex

```
1  public boolean AddVertex(int keyu){
2    while(true) {
3      VNode predu = head;
4      VNode curru = head.next;
5      while (curru.info.key < keyu){
6         predu = curru;
```

```
7         curru = curru.next;
8       }
9       predu.lock();
10      try {
11        curru.lock();
12        try {
13          if(validateVertex(predu, curru)) {
14            if(curru.info.key == keyu){/*present*/
15               return false;
16             } else {/*not present*/
17               ENode newnode = new ENode(u);
18               newnode.next = curru;
19               predu.next = newnode;
20               return true;
21             }
22          }
23        } finally{ /*always unlock*/
24            curru.unlock();
25         }
26        } finally {/*always unlock*/
27             predu.unlock();
28        }
29    }/*end of while(true)*/
30  }
```

Listing 10: Remove Vertex

```
1  public boolean RemoveVertex(int keyu){
2    while (true) {
3     VNode predu = head;
4     VNode curru = head.next;
5     while (curru.info.key < keyu){
6       predu = curru;
7       curru = curru.next;
8     }
9     predu.lock();
10    try {
11      curru.lock();
12      try{
13       if(validateVertex(predu, curru)){
14         if (curru.info.key != keyu){/*absent*/
15             return false;
16         }else {/*vertex keyu present*/
17            curru.marked = true;/*logical removal*/
18            predu.next = curru.next;/*physical removal of vertex*/
19    /*physical removal of edges to the vertex keyu*/
20             VNode temp = head;
21             while(temp != null) {
22                RemoveEdge(temp.info.key, keyu);
23                temp = temp.next;
24             }
25             return true;
26           }
27         }
28       }finally{/*always unlock currv*/
29             curru.unlock();
30      }
31     } finally {/*always unlock predv*/
32        predu.unlock();
33    }
34   }/*end of while(true)*/
35  }
```

Listing 11: Obstruction-Free Snapshot

```
1
2 import java.util.Arrays;
3 public NodeList collect(){/*get the snapshot of the graph*/
4   public NodeList snap_head = NULL;
5   public NodeList temp_snap, original, snap_new;
6   public Node snap_node, original_node, new_node;
7     original = graph;
8     while(original != NULL){
9      if(original.listhead.marked.load() == true)
10        break next1;
11     snap_new = new NodeList();
12     snap_new.listhead.key = original.listhead.key;
13     snap_new.listhead.next = NULL;
14     snap_new.listhead.marked.store(false);
15     snap_new.next = NULL;
16     if(snap_head == NULL){
17       temp_snap = snap_new;
18       snap_head = snap_new;
19     }
20     else{
21      temp_snap.next = snap_new;
22      temp_snap = snap_new;
23     }
24     original_node = original.listhead.next;
25     while(original_node != NULL){
26      if(original_node.marked.load() == true)
27        break next2;
28      new_node = new Node();
29      new_node.key = original_node.key;
30      new_node.marked.store(false);
31      new_node.next = NULL;
32      if(temp_snap.listhead.next==NULL){/*1^{st} node*/
33       temp_snap.listhead.next = new_node;
34       snap_node = new_node;
35      }
36      else{
37       snap_node.next = new_node;
38       snap_node = new_node;
39      }
40 next2:original_node = original_node.next;
41     }
42 next1:original = original.next;
43   }
44  return snap_head;
45 }
46 public NodeList scan(){/*get 2 consecutive same snapshot of the graph*/
47   public NodeList old_copy, new_copy, temp1, temp2;
48   public Node node_temp1, node_temp2;
49   old_copy = collect();
50 header: new_copy = collect();
51   temp1 = old_copy;
52   temp2 = new_copy;
53   while(temp1 != NULL && temp2 != NULL){
54     if(temp1.listhead.key == temp2.listhead.key){
55       node_temp1 = temp1.listhead.next;
56       node_temp2 = temp2.listhead.next;
57       while(node_temp2 != NULL && node_temp1 != NULL){
58        if(node_temp1.key == node_temp2.key){
59          node_temp1 = node_temp1->next;
60          node_temp2 = node_temp2->next;
61        }
62        else{
```

```
63          /*free the old_copy*/
64          old_copy = new_copy;
65          break header;
66         }
67       }
68       temp1 = temp1.next;
69       temp2 = temp2.next;
70     }
71     else{
72     /*free the old_copy*/
73      old_copy = new_copy;
74      break header;
75   }
76  }
77 return new_copy;
78 }
79
80 public boolean CycleDetect(){
81  public NodeList snapshot;
82  snapshot = scan();
83  boolean cycle = false;
84  cycle = call the DFS algorithm to check the Cycle on the snapshot
85   if cycle detected
86    call the RemoveEdge to delete the edge;
87    return true;
88   else
89    return false;
90 }
```

Listing 12: Wait-Free Reachability

```
1  public boolean Reach(int keyu, int keyv){
2    Set local_R ← φ/*Set does not allow duplicates*/
3    VNode curru = head;
4    /*traversing vertex list */
5    while (curru.info.key < keyu){
6        curru = curru.next;
7    }
8    if(curru.info.key != keyu || curru.info.marked)
9       return false;/*vertex keyu is not present, stop checking*/
10   ENode currv = curru.listhead.next;
11   while (currv.next != NULL){
12      if(!currv.info.marked)
13          local_R ← local_R ∪ currv.info.key
14      currv = currv.next;
15   }
16   if(keyv ∈ local_R)
17      return true;
18   end-if
19   Mark keyu as explored in local copy
20   for (each vertex keyw ∈ local_R which is not explored) {
21      VNode curr = head;
22      /*traversing vertex list */
23      while(curr.info.key < keyw){
24           curr = curr.next;
25      }
26      if(curr.info.key != keyw || curr.info.marked)
27           continue;
28      ENode currw = curr.listhead.next;
29      while (currw.next != NULL){
30        if(!currw.info.marked)
31             local_R ← local_R ∪ currw.info.key
32        currw = currw.next;
```

```
33        }
34        if(keyv ∈ local_R)
35           return true    /*reachable*/
36        end-if
37        Mark keyw as explored
38     }
39     return false; /*path does not exist from keyu to keyv*/
40  }
```