A Pragmatic Non-Blocking Concurrent Directed Acyclic Graph

Sathya Peri[1] Muktikanta Sa[1] Nandini Singhal[2]

[1]Department of Computer Science & Engineering,
Indian Institute of Technology Hyderabad, India
{sathya_p@iith.ac.in, cs15resch11012@iith.ac.in}
[2]Microsoft (R&D) Pvt. Ltd, Bangalore, India
nandini12396@gmail.com

Outline of the Presentation

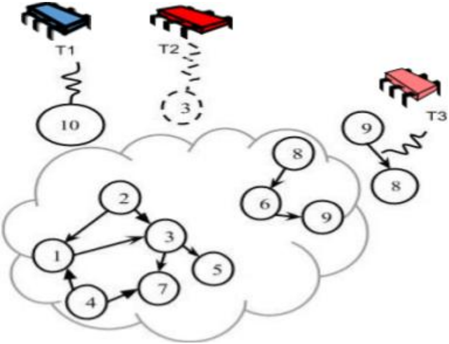
1. Introduction
2. The System Model
3. The ADT Operations
4. Modified ADT Operations: For Maintaining Acyclicity
5. The Data Structure
6. Acyclic Add Edge Operation
7. Reachability Methods to Test a Cycle
8. Correctness and Progress Guarantees
9. Simulation Results

Introduction

- Common real world objects can be modeled as graphs, which build the pairwise relations between objects.
- Graph algorithms applied in many applications, including social networks, communication networks, VLSI design, graphics, etc.
- Often these graphs are dynamic in nature and the updates are real-time.

Introduction

- Common real world objects can be modeled as graphs, which build the pairwise relations between objects.
- Graph algorithms applied in many applications, including social networks, communication networks, VLSI design, graphics, etc.
- Often these graphs are dynamic in nature and the updates are real-time.



Acyclic Graph

Several applications which maintain dynamic graphs require it to be acyclic such as:

- Databases: *Serialization Graph Testing (SGT)* in the field of Databases and Transactional Memory Systems (TM).
- Blockchains: Several blockchains maintain acyclic graphs such as tree structure (Bitcoin, Ethereum) or general DAGs (Tangle).
- Deadlock Detection: Several deadlock detection algorithms have been proposed in literature that require maintenance of acyclic graphs.
- Data processing, Data compression etc.

The System Model

- 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.
- Execution on a shared-memory multi-processor system which supports atomic `read`, `write`, `fetch-and-add (FAA)` `and` `compare-and-swap (CAS)` instructions.

The System Model

- 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.
- Execution on a shared-memory multi-processor system which supports atomic `read`, `write`, `fetch-and-add (FAA) and compare-and-swap (CAS)` instructions.

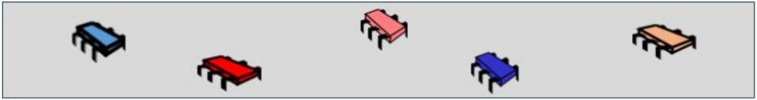
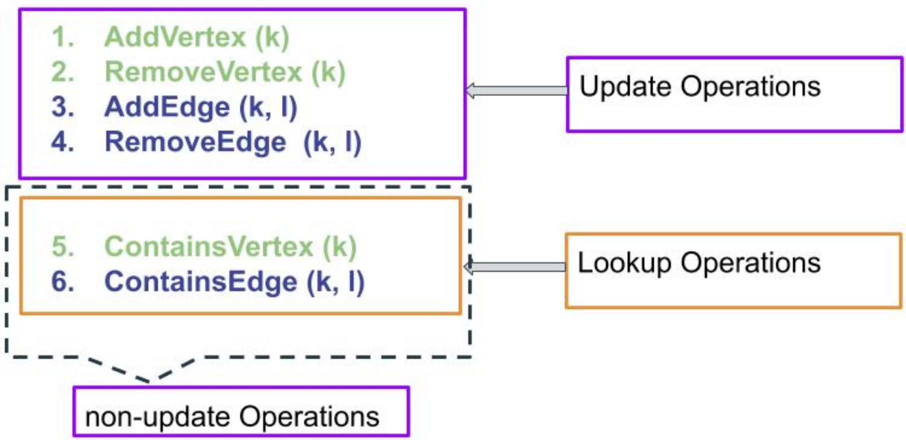


Figure: Concurrent Threads.

The ADT Operations [a]



1. AddVertex (k)
2. RemoveVertex (k)
3. AddEdge (k, l)
4. RemoveEdge (k, l)

Update Operations

5. ContainsVertex (k)
6. ContainsEdge (k, l)

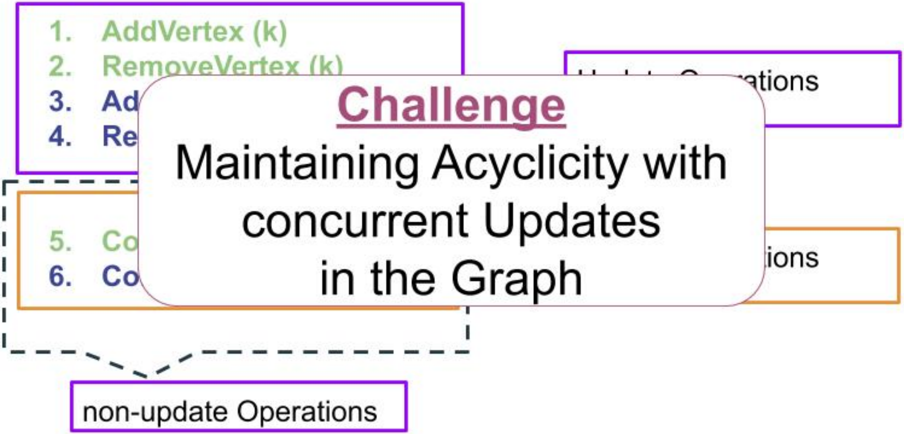
Lookup Operations

non-update Operations

---

[a] Bapi Chatterjee, Sathya Peri, Muktikanta Sa, and Nandini Singhal. *A Simple and Practical Concurrent Non-blocking Unbounded Graph with Linearizable Reachability Queries*,ICDCN 2019.

The ADT Operations



1. AddVertex (k)
2. RemoveVertex (k)
3. Ad
4. Re
5. Co
6. Co

ations

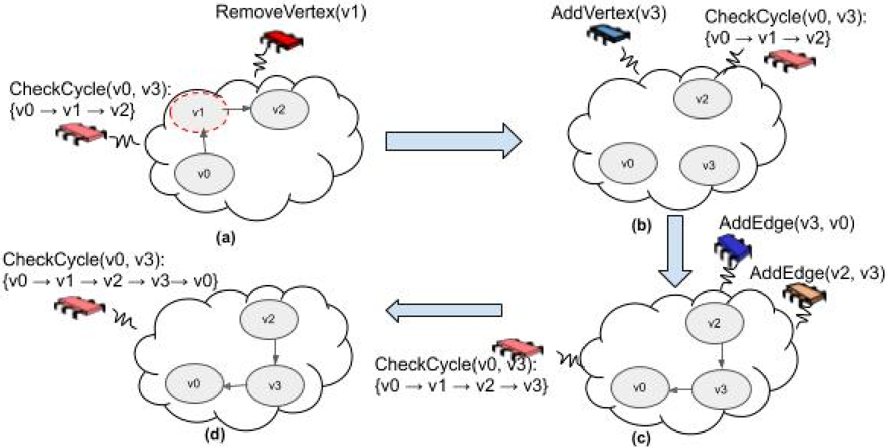
ions

non-update Operations

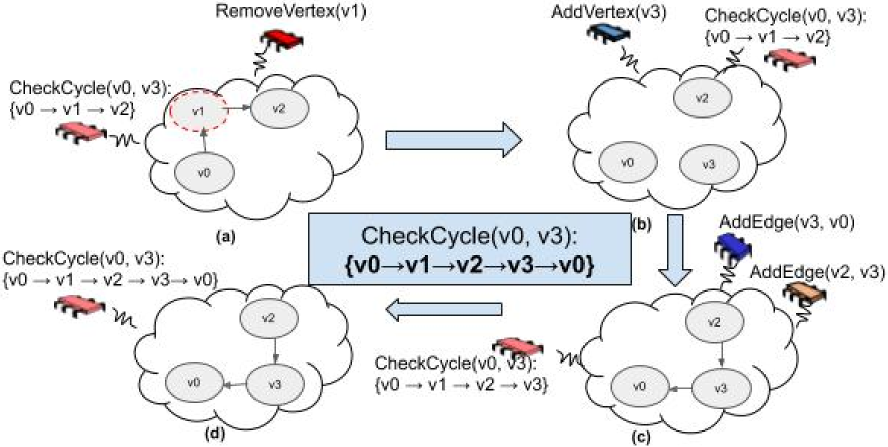
Challenge

Maintaining Acyclicity with concurrent Updates in the Graph

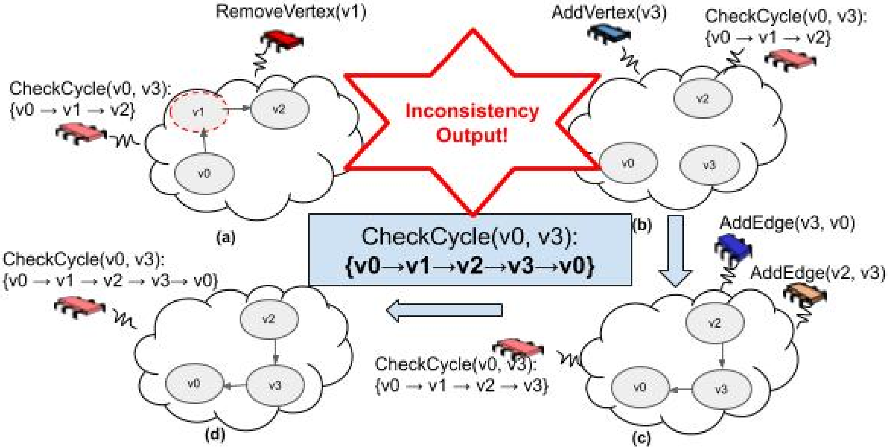
Difficulty with Maintaining Acyclicity



Difficulty with Maintaining Acyclicity



Difficulty with Maintaining Acyclicity



Correctness

Correctness

- The inconsistency is due to violation of correctness.

Correctness

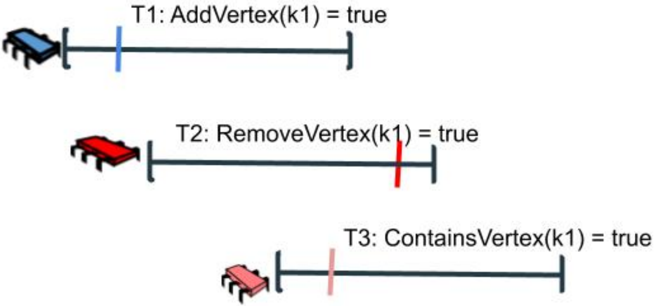
- The inconsistency is due to violation of correctness.
- The correctness-criterion that we consider is *linearizability*.

Correctness

- The inconsistency is due to violation of correctness.
- The correctness-criterion that we consider is *linearizability*.
- A concurrent data-strcture d is linearizable if for any history (execution) H output by d:
    - Assign an atomic step as a linearization point (LP) inside the execution interval of each of the operations.
    - The history H is equivalent to a valid sequential execution obtained by ordering the operations by their LPs.

Linearizability Example: Set Data-Structure

Linearizability Example: Set Data-Structure



Progress Guarantees

Progress Guarantees

Wait-free

A method is wait-free if it guarantees that every call finishes its execution in a finite number of steps.

Progress Guarantees

Wait-free

A method is wait-free if it guarantees that every call finishes its execution in a finite number of steps.

Lock-free

A method is lock-free if it guarantees that infinitely often some method call finishes in a finite number of steps.

Progress Guarantees

Wait-free

A method is wait-free if it guarantees that every call finishes its execution in a finite number of steps.

Lock-free

A method is lock-free if it guarantees that infinitely often some method call finishes in a finite number of steps.

Obstruction-free

A method is obstruction-free if, from any point after which it executes in isolation, it finishes in a finite number of steps (method call executes in isolation if no other threads take steps).

Modification of ADT Operations: For Maintaining Acyclicity

1. AddVertex
2. RemoveVertex
3. ContainsVertex
4. AddEdge
5. RemoveEdge
6. ContainsEdge

Modification of ADT Operations: For Maintaining Acyclicity

1. AddVertex
2. RemoveVertex
3. ContainsVertex
4. **AddEdge**
5. RemoveEdge
6. ContainsEdge

Modification of AddEdge Operation: For Maintaining Acyclicity

Modification of AddEdge Operation: For Maintaining Acyclicity

1. Only edge addition can cause cycle
2. So, we modify AddEdge to ensure that no cycles are formed.

The Data Structure

A directed graph $G = (V, E)$

The Data Structure

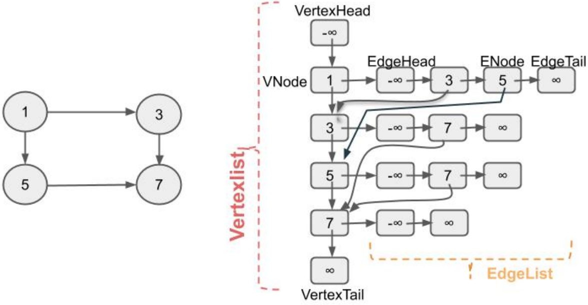
A directed graph $G = (V, E)$

- represented as an adjacency list
- enables it to grow (up to the availability of memory) and sink at the runtime.
- based on [Chatterjee et. al., ICDCN 2019]

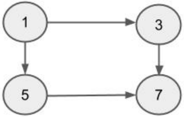
The Data Structure

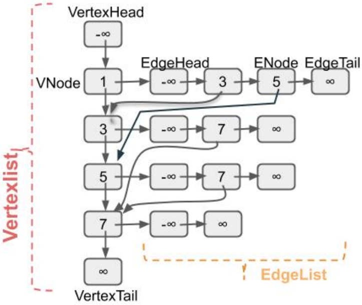
A directed graph $G = (V, E)$

- represented as an adjacency list
- enables it to grow (up to the availability of memory) and sink at the runtime.
- based on [Chatterjee et. al., ICDCN 2019]

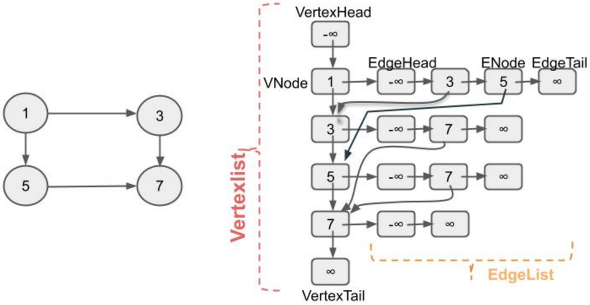


The Data Structure - Node States





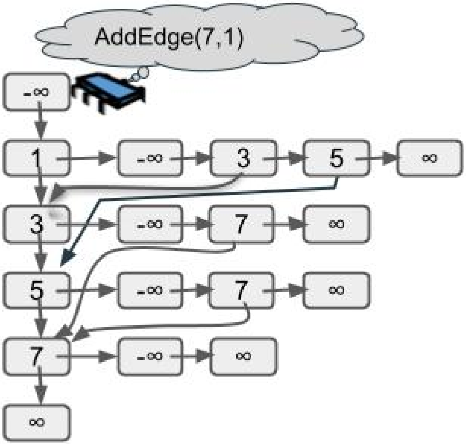
The Data Structure - Node States

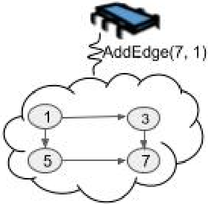


The states of the nodes

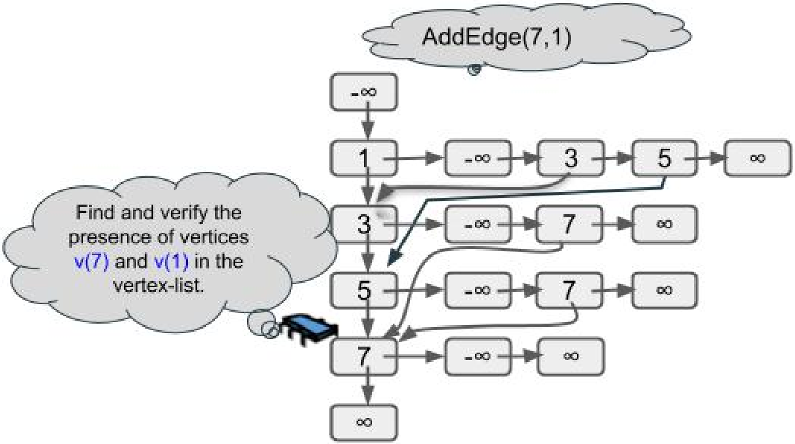
- VNode: `MARKED` or `UNMARKED`. Similar to a concurrent list-based set.
- ENode: `MARKED` or `ADDED` or `TRANSIT`.

Acyclic Add Edge Operation





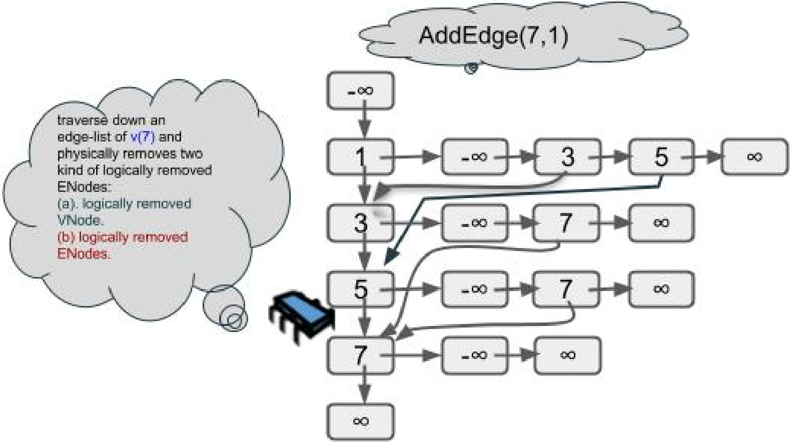
Acyclic Add Edge Operation



AddEdge(7,1)

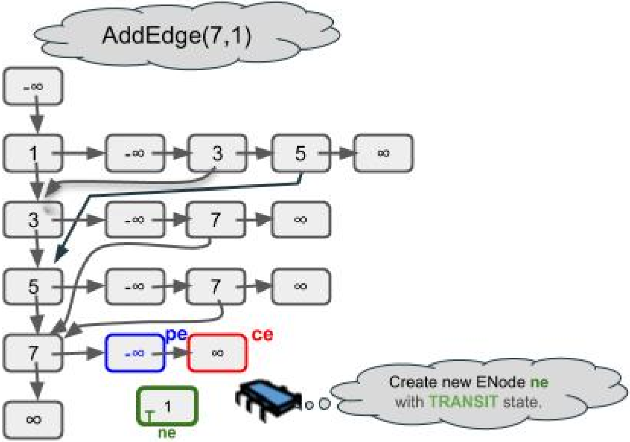
Find and verify the presence of vertices v(7) and v(1) in the vertex-list.

Acyclic Add Edge Operation Cont...

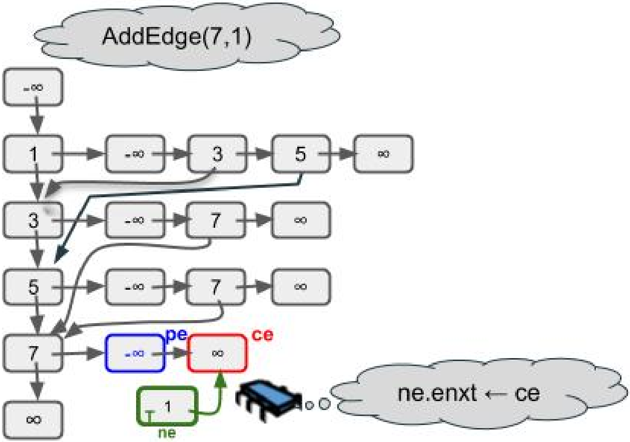


traverse down an edge-list of v(7) and physically removes two kind of logically removed ENodes:
(a). logically removed VNode.
(b) logically removed ENodes.

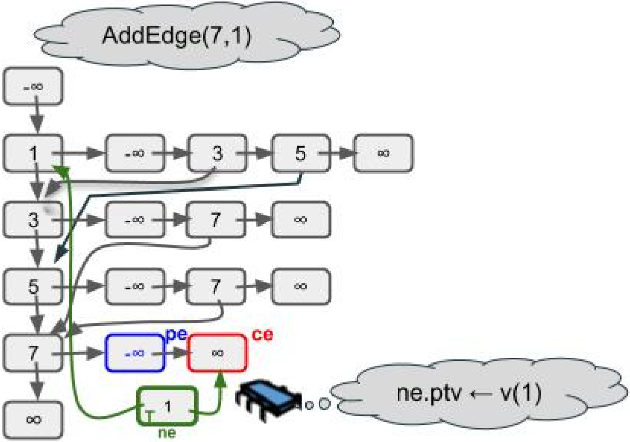
Acyclic Add Edge Operation Cont...



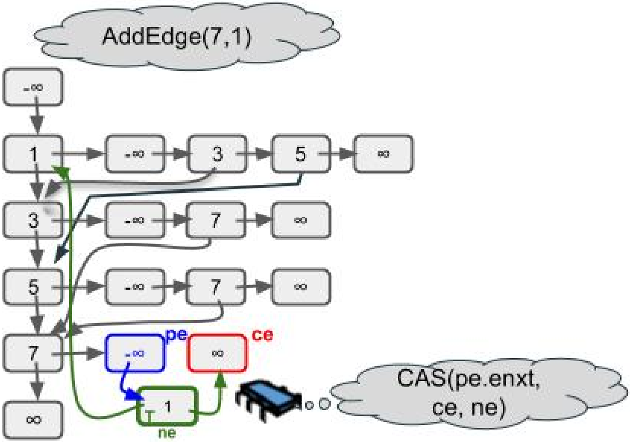
Acyclic Add Edge Operation Cont...



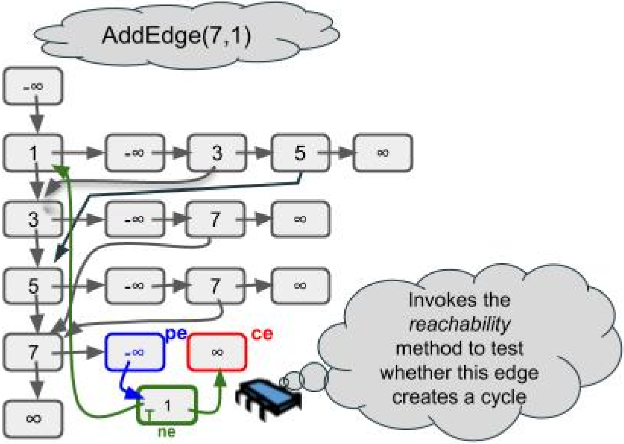
Acyclic Add Edge Operation Cont...



Acyclic Add Edge Operation Cont...



Acyclic Add Edge Operation Cont...



Algorithm of Add Edge

1. Add an edge with the `TRANSIT` state.

Algorithm of Add Edge

1. Add an edge with the `TRANSIT` state.
2. Invokes the *reachability* method to test whether this edge creates a cycle or not

Algorithm of Add Edge

1. Add an edge with the `TRANSIT` state.
2. Invokes the *reachability* method to test whether this edge creates a cycle or not
    1. If so, we delete the edge by setting its state from `TRANSIT` to `MARKED` and return `false` along with an indicative string `CYCLE DETECTED`.
    2. Otherwise, we set the state from `TRANSIT` to `ADDED` and return `true` along with an indicative string `EDGE ADDED`.

Algorithm of Add Edge

1. Add an edge with the `TRANSIT` state.
2. Invokes the *reachability* method to test whether this edge creates a cycle or not
    1. If so, we delete the edge by setting its state from `TRANSIT` to `MARKED` and return `false` along with an indicative string `CYCLE DETECTED`.
    2. Otherwise, we set the state from `TRANSIT` to `ADDED` and return `true` along with an indicative string `EDGE ADDED`.
3. An edge in `TRANSIT` state is not visible to containsEdge operations.

Reachability Methods to Test a Cycle

We present two approaches for maintaining acyclicity:

1. First one is based on a Wait-free Reachability query (SCR: Single Collect Reachable)

---

[b] Bapi Chatterjee, Sathya Peri, Muktikanta Sa, and Nandini Singhal. *A Simple and Practical Concurrent Non-blocking Unbounded Graph with Linearizable Reachability Queries*,ICDCN 2019.

Reachability Methods to Test a Cycle

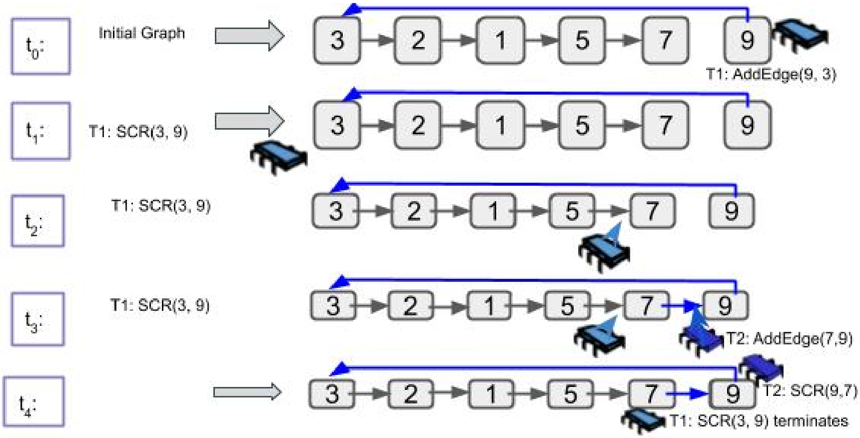
We present two approaches for maintaining acyclicity:

1. First one is based on a Wait-free Reachability query (SCR: Single Collect Reachable)
2. Second one is based on a Obstruction-free Reachability query (DCR: Double Collect Reachable), similar to the `GetPath`[b] algorithm.

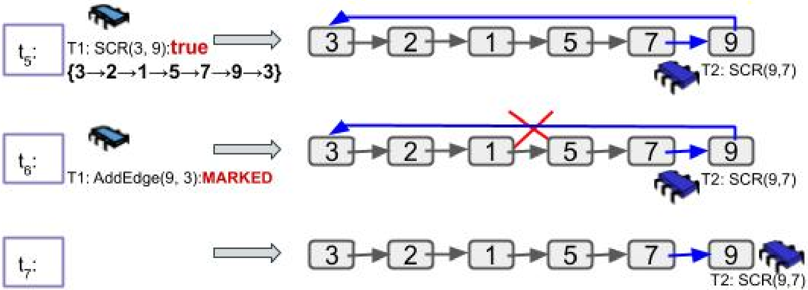
---

[b] Bapi Chatterjee, Sathya Peri, Muktikanta Sa, and Nandini Singhal. *A Simple and Practical Concurrent Non-blocking Unbounded Graph with Linearizable Reachability Queries*,ICDCN 2019.

SCR(k, l) : Wait-free Reachability

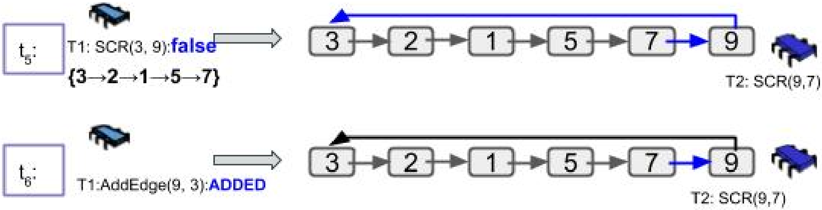


SCR(k, l) : Wait-free Reachability



OR

SCR(k, l) : Wait-free Reachability



SCR(k, l) : Wait-free Reachability

Algorithm

1. Performs non-recursive BFS traversal starting from the vertex k.

SCR(k, l) : Wait-free Reachability

Algorithm

1. Performs non-recursive BFS traversal starting from the vertex k.
2. Explores all VNodes which are reachable from k and are unmarked, i.e., are either `TRANSIT` or `ADDEDENodes`.

SCR(k, l) : Wait-free Reachability

Algorithm

1. Performs non-recursive BFS traversal starting from the vertex k.
2. Explores all VNodes which are reachable from k and are unmarked, i.e., are either `TRANSIT` or `ADDEDENodes`.
3. If it reaches l, then it terminates by returning `true` to the AddEdge operation.

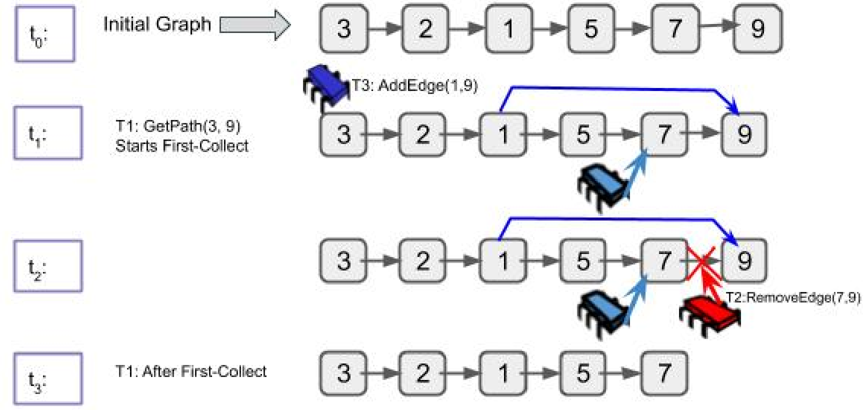
SCR(k, l) : Wait-free Reachability

Algorithm

1. Performs non-recursive BFS traversal starting from the vertex k.
2. Explores all VNodes which are reachable from k and are unmarked, i.e., are either `TRANSIT` or `ADDEDENodes`.
3. If it reaches l, then it terminates by returning `true` to the AddEdge operation.
4. If it unable to reach l then it terminates by returning `false` to the AddEdge operation.

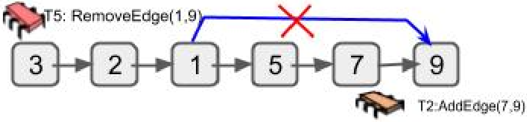
DCR(k,l) : Obstruction-free Reachability

Double Collect Problem: First Collect

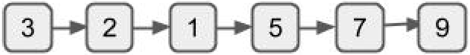


After First Collect Graph Restored

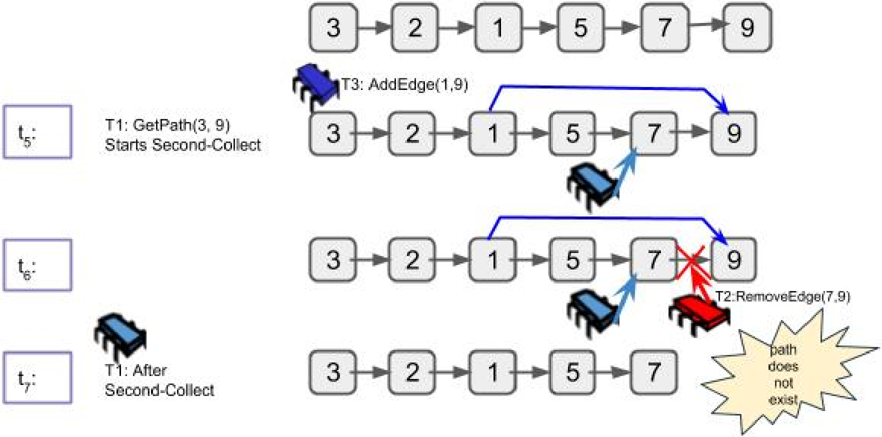
t_4:



Graph has been restored



Double Collect Problem: Second Collect



Solution

To solve these issues...

1. We take double collect.
2. In each scan we collect BFS-tree which is a partial snapshot.

Solution

To solve these issues...

1. We take double collect.
2. In each scan we collect BFS-tree which is a partial snapshot.
3. To capture the modifications.
    1. We have a counter associated with each vertex.
    2. Whenever any edge operations happens the counter incremented.

Solution

To solve these issues...

1. We take double collect.
2. In each scan we collect BFS-tree which is a partial snapshot.
3. To capture the modifications.
    1. We have a counter associated with each vertex.
    2. Whenever any edge operations happens the counter incremented.
4. To verify the double collect we compare with BFS-tree alone with counter.
5. If the both the double collects are same
    1. We have valid snapshot
    2. We analyse the the valid snapshot for the presence or absence of the path.

Correctness and Progress Guarantees

Theorem 1:

1. The ADT operations are linearizable.

Correctness and Progress Guarantees

Theorem 1:

1. The ADT operations are linearizable.

Theorem 2:

The ADT operations are non-blocking:

1. The operations *ContainsVertex*, *ContainsEdge* and *SCR* are wait-free.
2. The operation *DCR* is obstruction-free.
3. The operations *AddVertex*, *RemoveVertex*, *ContainsVertex*, *AddEdge*, *RemoveEdge*, and *ContainsEdge* are lock-free.

Correctness and Progress Guarantees

Theorem 1:

1. The ADT operations are linearizable.

Theorem 2:

The ADT operations are non-blocking:

1. The operations *ContainsVertex*, *ContainsEdge* and *SCR* are wait-free.
2. The operation *DCR* is obstruction-free.
3. The operations *AddVertex*, *RemoveVertex*, *ContainsVertex*, *AddEdge*, *RemoveEdge*, and *ContainsEdge* are lock-free.

Proofs of the Theorem 1 and 2 are shown in the paper.

Experimental Setup

Experimental Setup

- Intel(R) Xeon(R) E5-2690 v4 CPU containing 14 cores running at 2.60GHz on two sockets. Each core supports 2 logical threads.
  - Thus, a total of 56 logical cores.
- Implementation in C++ without any garbage collection. Multi-threaded implementation is based on Posix threads.

Experimental Setup

- Intel(R) Xeon(R) E5-2690 v4 CPU containing 14 cores running at 2.60GHz on two sockets. Each core supports 2 logical threads.
    - Thus, a total of 56 logical cores.
- Implementation in C++ without any garbage collection. Multi-threaded implementation is based on Posix threads.
- Start experiments, with 1000 vertices and approximately 125000 edges added randomly.

Experimental Setup

- Intel(R) Xeon(R) E5-2690 v4 CPU containing 14 cores running at 2.60GHz on two sockets. Each core supports 2 logical threads.
    - Thus, a total of 56 logical cores.
- Implementation in C++ without any garbage collection. Multi-threaded implementation is based on Posix threads.
- Start experiments, with 1000 vertices and approximately 125000 edges added randomly.
- We measure throughput obtained on running the experiment for 20 seconds.
- Each data point is obtained by averaging over 7 iterations.

Experimental Setup

- Intel(R) Xeon(R) E5-2690 v4 CPU containing 14 cores running at 2.60GHz on two sockets. Each core supports 2 logical threads.
  - Thus, a total of 56 logical cores.
- Implementation in C++ without any garbage collection. Multi-threaded implementation is based on Posix threads.
- Start experiments, with 1000 vertices and approximately 125000 edges added randomly.
- We measure throughput obtained on running the experiment for 20 seconds.
- Each data point is obtained by averaging over 7 iterations.
- We compare the SCR and DCR with its sequential and coarse-grained counterparts.

Workload Distributions

Graph Operations: AddVertex, RemoveVertex, ContainsVertex, AddEdge, RemoveEdge and ContainsEdge

- **Lookup Intensive:** (2.5%, 2.5%, 45%, 2.5%, 2.5%, 45%)
- **Equal Lookup and Updates:** (12.5%, 12.5%, 25%,12.5%, 12.5%, 25%)
- **Update Intensive:** (22.5%, 22.5%, 5%, 22.5%, 22.5%, 5%)

Workload Distributions

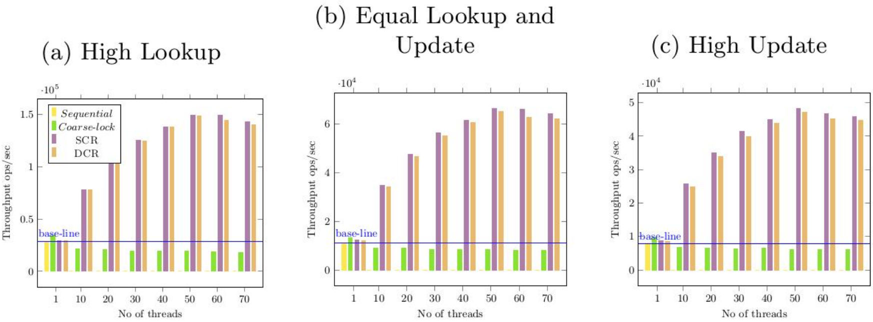
Graph Operations: AddVertex, RemoveVertex, ContainsVertex, AddEdge, RemoveEdge and ContainsEdge

- **Lookup Intensive:** (2.5%, 2.5%, 45%, 2.5%, 2.5%, 45%)
- **Equal Lookup and Updates:** (12.5%, 12.5%, 25%,12.5%, 12.5%, 25%)
- **Update Intensive:** (22.5%, 22.5%, 5%, 22.5%, 22.5%, 5%)

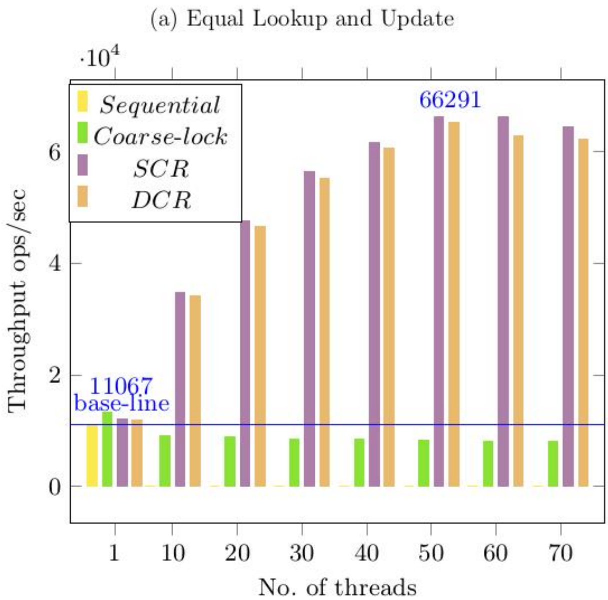
We have compared the following cases.

S. No	Label	Explanation
1	Sequential	Sequential execution of all the operations
2	Coarse-lock	Coarse lock execution of all the operations
3	SCR	AddEdge based on Single Collect Reachable Algorithm
4	DCR	AddEdge based on Double Collect Reachable Algorithm

Results


(a) High Lookup
(b) Equal Lookup and Update
(c) High Update
·10^5
·10^4
·10^4
Sequential
Coarse-lock
SCR
DCR
base-line
Throughput ops/sec
No of threads
1
10
20
30
40
50
60
70
0
0.5
1
1.5
0
2
4
6
0
1
2
3
4
5

Results


(a) Equal Lookup and Update
$\cdot 10^4$
Sequential
Coarse-lock
SCR
DCR
66291
11067
base-line
Throughput ops/sec
0
2
4
6
1
10
20
30
40
50
60
70
No. of threads

Observation

1. Both SCR and DCR algorithms suffer from false-positives due to concurrent addition of edges.

Observation

1. Both SCR and DCR algorithms suffer from false-positives due to concurrent addition of edges.
2. In the future, we plan to measure the number of *false positives* incurred by these algorithms.

Observation

1. Both SCR and DCR algorithms suffer from false-positives due to concurrent addition of edges.
2. In the future, we plan to measure the number of *false positives* incurred by these algorithms.
3. Identify ways to reduce them.

Conclusion

Conclusion

1. Two efficient non-blocking concurrent algorithms for maintaining acyclicity in a directed graph where vertices and edges are dynamically inserted and/or deleted.

Conclusion

1. Two efficient non-blocking concurrent algorithms for maintaining acyclicity in a directed graph where vertices and edges are dynamically inserted and/or deleted.
2. The first algorithm is based on a wait-free reachability query: SCR
3. the second one is based on partial snapshot-based obstruction-free reachability query: DCR

Conclusion

1. Two efficient non-blocking concurrent algorithms for maintaining acyclicity in a directed graph where vertices and edges are dynamically inserted and/or deleted.
2. The first algorithm is based on a wait-free reachability query: SCR
3. the second one is based on partial snapshot-based obstruction-free reachability query: DCR
4. We extensively evaluate a sample C/C++ implementation of the algorithm through a number of micro-benchmarks.

Conclusion

1. Two efficient non-blocking concurrent algorithms for maintaining acyclicity in a directed graph where vertices and edges are dynamically inserted and/or deleted.
2. The first algorithm is based on a wait-free reachability query: SCR
3. the second one is based on partial snapshot-based obstruction-free reachability query: DCR
4. We extensively evaluate a sample C/C++ implementation of the algorithm through a number of micro-benchmarks.
5. Our experiments show that the proposed algorithm scales 7X with the number of threads in commonly available multi-core systems.

For More Information

1. The Technical Report is available at: *https://arxiv.org/abs/1611.03947*
2. And the complete source code is available at: *https://github.com/PDCRL/ConcurrentGraphDS*

Thank You!

For Further Reading..

- Chatterjee B. et al. *A Simple and Practical Concurrent Non-blocking Unbounded Graph with Linearizable Reachability Queries*. Proceedings of the 20th International Conference on Distributed Computing and Networking, ICDCN 2019
- Maurice P. et al. *Linearizability: A Correctness Condition for Concurrent Objects*. ACM Transactions on Programming Languages and Systems, Vol. 12, No. 3, July 1990, Pages 463-492.
- Y. Riany. et al. *Towards a practical snapshot algorithm*. Theoretical Computer Science, 269(1-2): 163-201, 2001.
- Timothy L. Harris. *A Pragmatic Implementation of Non-blocking Linked-Lists*. Distributed Computing, 15th International Conference, DISC 2001.
- Maurice Herlihy and Nir Shavit. *The Art of Multiprocesor Programming, Revised Print*. Imprinted Morgan Kaufmann,Elsevier, May 2012.
- A. Natarajan and N. Mittal, *Fast concurrent lock-free binary search trees* 19th PPoPP, 2014, pp. 317–328.
- Arnab Sinha, Sharad Malik, *Runtime checking of serializability in software transactional memory*, Parallel & Distributed Processing (IPDPS), 2010
- Khanh Do Ba, *Wait-Free and Obstruction-Free Snapshot*, Dartmouth Computer Science Technical Report TR2006-578, June 2006.
- Y. Afek, H. Attiya, D. Dolev, E. Gafni, M. Merritt and N. Shavit. *Atomic snapshots of shared memory*. Proc. ACM PODC , 1–14, 1990.
- Steve Heller, Maurice Herlihy, Victor Luchangco, Mark Moir, William N. Scherer III, Nir Shavit. *A Lazy Concurrent List-Based Set Algorithm*. Parallel Processing Letters, volume 17, 4, 411–424, 2007,