An On-line Algorithm for Maintaining the Transitive Closure in Shared-memory Graph

Sathya Peri and Muktikanta Sa

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

Abstract. In this paper, we present an on-line full dynamic algorithm for maintaining transitive closure of a directed graph in a shared memory architecture. The edges are added or deleted and vertices are added concurrently by fixed number of threads. To the the best of our knowledge, this is the first work to propose using linearizable concurrent directed graph and is build using ordered list-based set. We provide an empirical comparison against sequential, course-grained and hoh-locking. The results show our algorithm perform well an increasing number of threads. The throughput is increased between 3-6x depending on different workload distributions and applications. We believe that there are huge applications in the on-line graph. Finally, we show how the algorithm can be extended to descendant counting problem in on-line graph.

Keywords: concurrent data structure; directed graph; transitive closure, locks; connectivity on directed graphs; dynamic graph algorithms;

1 Introduction

Generally the real-world practical graph always dynamically change over time. Dynamic graphs are the one's which are subjected to a sequence of changes like insertion, deletion of vertices and/or edges [4]. Dynamic graph algorithms are used extensively and it has been studied for several decades. Many important results have been achieved for fundamental dynamic graph problems and some of these problems are very challenging i.e, transitive closure, finding cycles, graph coloring, minimum spanning tree, shortest path between a pair of vertices, connectivity, 2-edge & 2-vertex connectivity, strongly connected components, flow network, etc (see, e.g., the survey in [4]). The social, biological type of graph networks are very complicated due to their complex layered architecture and size. Graphs Networks, typically involving finding the perfect match or similarities on gene expression for evolution like disease progression, the shortest(reachable) path between pair of proteins, [21] are even more challenging to the research community.

We have been specifically motivated by largely used problem of fully dynamic evolution *Transitive Closure*(TC). Finding TC in dynamically changing graph affects a large community both in the theoretical computer science and the network community. TC finding on static networks fails to capture the natural phenomena and important dynamics. Discovering TCs on dynamic graph helps uncover the laws in processes of graph evolution, which have been proven necessary to capture essential structural information in on-line social networking platforms (facebook, linkedin, google+, twitter, quora, etc.). TC often merges or splits because of the changing friendship over time. A common application of TC on these social graph is to check weather two members are reachable to each other. So, for a transitive closure graph(TC-graph) we define the *TC(id1,id2)*: which checks if there is a directed path from $id1$ to $id1$ and not the other way round. In general, a social network graph handles the concurrency control over a set of users or threads running concurrently. A thread as a block of code is invoked by the help of methods to access multiple shared memory objects atomically.

In this paper, we present a new shared-memory algorithm called as *SMTC* for maintaining transitive closure in fully dynamic directed acyclic graphs.The following are the key contributions of this work:

1. Firstly, we designed an incremental algorithm(SMITC) for maintaining TC dynamically. i.e, after inserting an edge or a vertex how quickly we update the TC-graph.
2. Secondly, we designed a decremental algorithm(SMDTC) for maintaining TC dynamically. i.e, after deleting an edge how quickly we update the TC-graph, we assume no deletion of the vertices.

3. Thirdly, an algorithm for maintaining fully dynamic transitive closure (SMTC).
4. An empirical comparison against sequential, course-grained [10, Ch 9] and hoh-locking [10, Ch 9].
5. Our algorithm is work-efficient for most on-line graphs.
6. An application suite : descendant counting problem on-line graph.

We have not found any comparable concurrent data-structure for solving this transitive closure problem in shared-memory architecture. Hence we crosscheck against sequential, course-grained [10, Ch 9] and hoh-locking [10, Ch 9] implementations. We found the algorithm proposed by Demetrescu & Italiano's [5] is not worthier of comparison, as they use pointer matrix($n \times n$) for fixed n set of vertices.

1.1 Background and Related Work

Let a concurrent directed graph $G = (V, E)$, $G(V)$ is a set of vertices and set of $G(E)$ directed edges. We use *Adj(u)* to denote the set of neighbors of a vertex u. The $G(E)$ is collection of both outgoing and incoming neighbors, i.e., *Adj(u)* $=\{$for outgoing edges $v : \langle u, v\rangle$ & for incoming edges $w : \langle w, u\rangle \in G(E)\}$. Each edge connects an ordered pair of vertices currently belongs to $G(V)$. And this G is dynamically being modified by a fixed set of concurrent running threads. Our dynamic graph setting, threads can perform insertion/deletion of edges and insertion of vertices. We assume that all the vertices have unique identification key, which is captured by *val* field in `Vnode`, `Enode` and `TCnode` structure which is shown in the Section 4.

Definition 1. (Reachability)*, Given a graph $G = (V, E)$ and two vertices $u, v \in G(V)$, a vertex v is reachable from another vertex u if there is a path from u to v.*

Definition 2. (Transitive Closure)*, Let $G = (V, E)$ be a directed graph, We say G' a transitive closure of a graph, if a vertex u is reachable from another vertex v for all vertex pairs (u, v). Here reachable mean that there exist a path from vertex u to v. The reachability matrix is called transitive closure of a graph G.*

Apart from the definition, transitive closure also satisfies the partial order relation on the set of vertices in directed acyclic graph:

1. *Reflexive:* every vertex v is transitive closure to itself.
2. *Antisymmetric:* if u is transitive closure to v, then v is not transitive closure to u.
3. *Transitive:* if u is transitive closure to v and v is transitive closure to w, then u is also transitive closure to w.

Related Work:There have been many parallel computing algorithms proposed for computing TC both in directed and undirected graphs. Generally the problems are related to application specific in on-line graph network what exactly we are trying to achieve. The main objective is, after any dynamic update operation, how quickly we restore the TC-graph instate starting everything from scratch and avoid recomputations. This helps to achieve the dynamic queries like reachability [17], shortest path [14, 17], connectivity [18, 20] etc. as quick as possible.

Henzinger and King [8, 9] proposed decremental reachability algorithm and they used spanning tree for each reachable vertices, which helps them to update the data-structure quickly only when deletion of edge occurs. Also in 2001 Frigioni et al. [6] developed a new deterministic algorithm to find the recability with faster update time.

Roditty et al. [16] proposed a dynamic reachability algorithms for directed graphs only deletion of an arbitrary set of edges. They proposed a decremental data structure with recability tree called incoming *In(w)* and outgoing *Out(w)* trees for each vertex w. First they delete the edges from the decremental data structure and then rebuild the In(w) and Out(w) tree from scratch with the number of phases.

The data-structure and algorithm developed by Demetrescu and Italiano [5] is well know for any update operation. Their dynamic reachablity data-structure is based on the matrix structure, with recursive decomposition in Kleene Closures. They maintain dynamic transitive closure of a graph in $O(n^2)$ amortized time per update like King's [13] algorithm. their algorithm supports batches of insertion and deletion on edges. Recently Bender et. al, [2] proposed incremental algorithm to maintain thetransitive closure of a dynamic graph.

In 2014, Slota, et. al., proposed a parallel multistep based algorithm using both BFS and coloring technique to detect the transitive closure and SCC in large graphs. Later they used the trimming methodology to reduce the search space of the graph to achieve better performance.

Bader. et. al., [1] developed a data-structure known as STRINGER for dynamic graph problems. They used combination of both adjacency matrices and Compressed Sparse Row (CSR) representation of graph. And they claimed that the STRINGER helps faster insertions and better spatio-temporal locality as compare to the adjacency lists representations. Later they also developed a CUDA version of the STRINGER called that cuSTRINGER [7], which supports dynamic graph algorithms for GPUs.

The main drawback of these algorithms is, they are expensive and need more space due to matrix use. None of above proposed algorithms clarify how the internal share-memory access is achieved by the multi-threads/processors and how the memory is synchronized, whether the data-structure is linealizable or not, etc. In this paper we able to address these problems.

The rest of the paper is organized as follows. In the Section 2, we define the system model, preliminaries and design principles. We define the data-structure of TC-graph in Section 4 and in the Section 5 we define technical details of all our algorithms and the pseudo-codes. In the Section 6 we give high level correctness proof and in the Section 7 we analyze the experimental results. Finally we concluded in the Section 8 along with future direction and discussion.

2 System Model & Preliminaries

In this paper, we have considered that our system consists of set of p processors, accessed by a finite set of n threads T_1, T_2 ...., T_n that run in a completely asynchronous manner and communicate through shared objects on which they perform atomic *read*, *write*, *fetch-and-add*(FAA) operations. A FAA operation takes two arguments (`loc, incVal`), where `loc` is the address location from where it fetches the value, then adds `incVal` to it and then writes back to the result `loc`.

We assume that each thread has a unique identifier, it is assigned at the time of thread creation. Each thread invokes a method which may be composed of shared-memory objects and local cipherings. We make no assumptions about the relative speeds of the threads and assume none of these processors and threads fails.

As we said earlier our proposed algorithms are implementations of shared objects and a share object is an abstraction of set of methods defined as `TC` class in the Section 4. It has set of methods and each method has its sequential specification. To prove a concurrent data structure to be correct, *linearizability* proposed by Herlihy & Wing [12] is the standard correctness criterion. A history is a sequence of invocations and responses made of an object by a set of threads. And each invocation of a method will have a subsequent response. The linearizability is defined as, each method call should appear to "take effect" instantaneously at some moment between its invocation and response [12]. Anytime a thread invokes a method for an object, it follows until it receives a response. It may be the case a method's invocation is pending, if it has not received a response. For any sequential history in which the methods are ordered by their linearization points(LPs).

Progress: An execution is *deadlock-free* if it guarantees minimal progress in every *crash-free* [11] execution, and maximal progress if it is starvation-free. An execution is *crash-free* if it guarantees minimal progress in every uniformly isolating history, and maximal progress in some such history [11].

Design Principles: We developed a set of correct behaviour for our algorithm and implementation.

1. *thread-safety*: The TC-graph data-structure can be shared by fixed number of multiple threads at all times, which ensures all fulfill their requirement specifications and behave properly without unintended interaction.
2. *lock-freedom*: Apply non-blocking techniques to provide an implementation of thread-safe C++ dynamic array based on the current C++ memory model.
3. *portability*: Generally our algorithms do not rely on specific hardware architectures, rather it is based on asynchronous memory model.
4. *simplicity*: The algorithm keeps the implementation simple to allow the correctness verification, like linearizability or model-based testing.

Notations: We denoted ↓, ↑ as input and output arguments to each method respectively and our pseudo-code is mixed of C/C++ and JAVA language format.

3 An Overview of the Algorithm

Before getting into the technical details of the algorithm, we first provide an overview of the design. The TC class supports some basic operations: AddVertex, AddEdge, DeleteEdge, checkDescendant, countDescendants, etc. First four methods are deadlock-free free and last two methods are wait-free. The high-level overview of the AddEdge and DeleteEdge methods are given bellow and the technical details are in the Section 5.

AddEdge (u, v):

1. First checks the presence of vertex u and v in the TC-graph. If both are present and edge is not present, adds v in the u's edge-list and adds -u in the v's edge-list. Else returns false.
2. After adding the edge successful, checks any changes to the TC-graph, if it is, invoke the *updateTCAfterAddE ()* method to restore the transitive closure.
3. First it adds the v & all its descendants(if present) to u's tc-list. Then backtracks all the vertices reachable to u and then updates the tc-list of all these vertices.

RemoveEdge (u, v):

1. First checks the presence of vertex u and v in the TC-graph. If both are present and edge is not present, deletes v from the u's edge-list and deletes -u from the v's edge-list. Else returns false.
2. After deleting the edge successful, checks any changes to the TC-graph, if it is, invoke the *updateTCAfterDelE ()* method to restore the transitive closure.
3. First it deletes the v & all its descendants(if present) from the u's tc-list. Then backtracks all the vertices reachable to u and then updates the tc-list of all these vertices.

4 The Underlying Data-Structure

In this section, we give a detailed construction of data-structure and it is depicted in the Fig 3. We represent it using linked-list of linked-lists the usual adjacency list representation of the graph with some modifications. All the vertices are stored in the vertex-list and it's neighbours are sorted in it's edge-list and the vertices which are reachable are stored in the tc-list. All the linked-lists are build from a set of totally-ordered *keys*(lower to higher order).

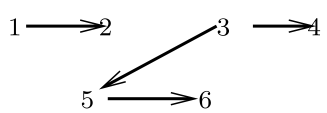

Fig. 1: An example of detected graph.

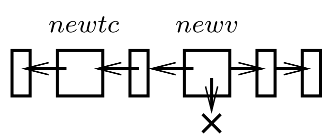

Fig. 2: Structure of new vertex(`newv`) and its new transitive closure `newtc`.

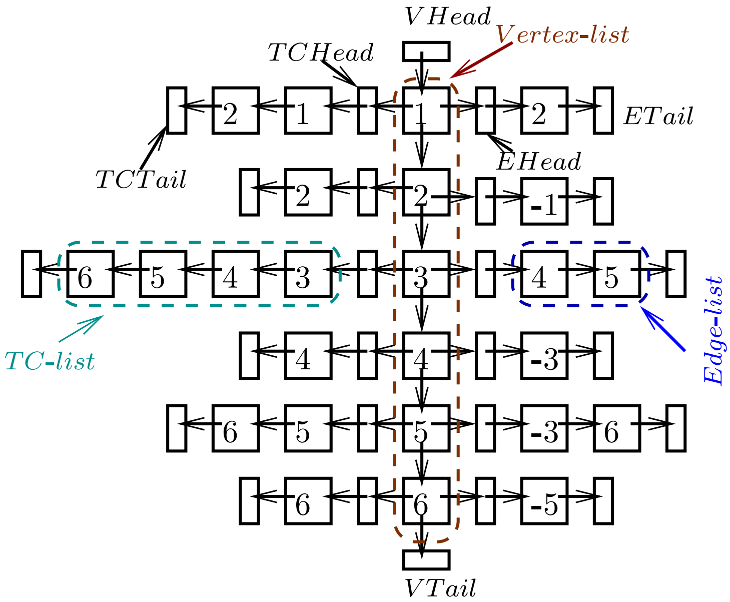

Fig. 3: The TC-graph, it the representation of the directed graph shown in Fig 1

The `Enode` structure is a normal node and has four fields. The `val` field is the actual value of the node. We represent all incoming edges with negative sign followed by key value `val` and outgoing edges with the key value `val`, as shown in the Fig 3. The main idea of storing both incoming and outgoing edges for each vertex helps to explore the graph backward and forward manner respectively. And also it helps to trim the TC-Graph after deleting a vertex, i.e, once a thread successfully deleted a vertex all its incoming and outgoing edges need to be removed quickly instate of iterating over whole TC-Graph. The boolean `marked` field is used to set the node and helps traversal to the target node without lock, we maintain an invariant that every unmarked node is reachable from the sentinel node `Head`. If a node is marked, then that is not logically present in the list. Each node has a `lock` field, that helps to achieve the *fine-grained* concurrency. Each node can be locked by invoking `lock()` and `unlock()` methods. It is just a fine-grained locking technique, helps multiple threads can traverse the list concurrently. The `enext` field is the atomic references to the next edge node in the edge-list.

```
struct Enode{
     long val;
     bool marked;
     Lock lock;
     Enode enext;
}elist_t;

struct TCnode{
     long val;
     bool marked;
     Lock lock;
     TCnode tcnext;
}tclist_t;

struct Vnode{
     long val;
     bool marked;
     Lock lock;
     elist_t enext;
     tclist_t tcnext;
     Vnode vnext;
}vlist_t;

class TC{
    vlist_t VHead,VTail;
    bool AddVertex(u);
    bool AddEdge(u,v);
    bool DeleteEdge(u,v);
    bool checkDesdnt(u,v);
    int  countDesdnt(u);
};
```

The `TCnode` structure is also a normal node like edge node and has four fields. The `val` field is the actual value of the node. The boolean `marked` and `lock` fields have same meaning as `Enode`. The `tcnext` field is the atomic references to the next transitive closure node in the tc-list.

The `Vnode` structure is used for holding all vertices belonging to a TC-graph. Like `Enode`, it has six fields. The `val` field is the actual key value of the vertex and it is unique. Once a key assigned to a vertex, same key will never generate again. We assume our system provides infinite number of unique keys and has no upper bound. The boolean `marked` and `lock` fields have same meaning as `Enode`. The `vnext` field is the atomic references to the next vertex node in the vertex-list. The `enext` and `tcnext` fields are the atomic references to edge-head(EH) and tc-head(TH) respectively. We also assume any graph can be handled with upper-bound provided by the current architecture.

Finally the `TC` class is the actual abstract class, which coordinates all operation activities. It has two sentinel nodes `VHead` and `VTail`. The *TC* class supports three basic graph operations *AddVertex*, *AddEdge* and *DeleteEdge*, and also supports some application specific methods, *checkDescendant*, *countDescendants*, etc. The detail working and pseudo code is given in the next section. When ever a sentinel created it assigned with -∞ and +∞ for `Head` and `Tail` respectively.

5 Algorithms

In this section we present SMTC, the actual algorithm for maintaining transitive closures of fully dynamic directed graph in a shared memory system. The edges and vertices are added/removed concurrently by fixed set of threads. The technical details of all the methods are discussed here. The basic transitive closure algorithm support the operations: *AddVertex (u)*, *AddEdge (u, v)*, *DeleteEdge (u, v)*, *checkDescendant (u, v)* and *countDescendants (u)*. In Section 3 we discussed the high level overview of two methods.

5.1 Incremental Algorithms(SMITC)

We say that an algorithm is incremental if it allows only insertions, it may be only edges insertion or only vertices insertion or both. Inserting an edge may cause hardship to restore the transitive closure properties, as reachable path may get affected. After any update operations executed successfully, how quickly we restore the TC-graph instate of starting everything from the scratch is not trivial. For that we proposed an incremental algorithm to maintain the transitive closure, we called it as SMITC algorithm and the details of the algorithm is given bellow. We used the modified data-structure proposed by Demetrescu & Italiano's [5] and backward depth-first search(BDFS) to find all reachable vertices to restore the affected TC after inserting an edge iff it violates the TC-graph. Whereas AddVertex will not affect the TC-graph, as it creates a new vertex without any neighbours.

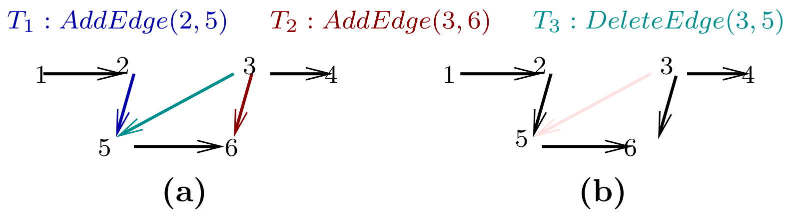

Fig. 4: (a)An example, T_1 & T_2 are adding edges (2,5) & (3,6) respectively and T_2 deleting the edge (3,5) on the graph shown Fig 1. (b). The graph after T_1, T_2 and T_3 successful performed operations. The corresponding TC-graph depicted in Fig. 5.

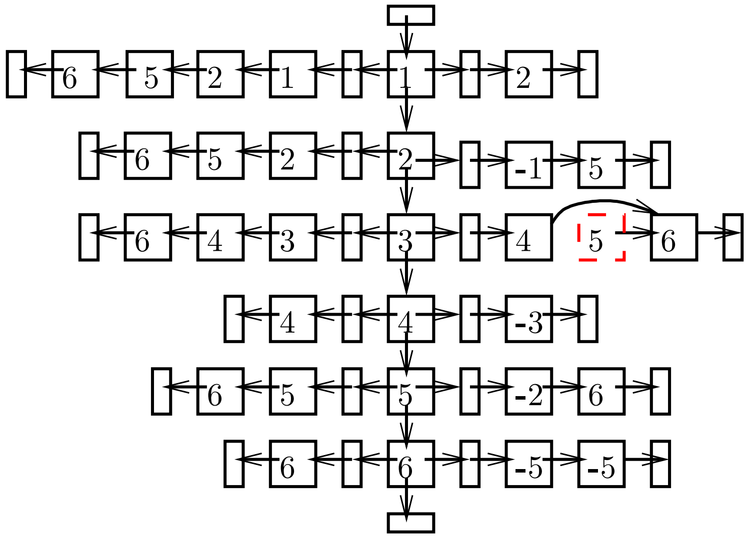

Fig. 5: The corresponding TC-graph of Fig 4

To add an edge, we invoke the $AddEdge(u, v)$ method presented in the Algorithm 15. First it checks the presence of vertices u and v by invoking the *Find ()* method (Algorithm 5) from Line 154 to 162. If any one of these vertices is not present or the edge is present, we simply return false. After successful check of u and v, in the the Line 163 we try to add the edge node v(outgoing edge) in the u's edge-list and the edge node -u(incoming edge) in the v's edge-list. After successful addition of both the edges, we add v to the u's tc-list (Line 165) by invoking the generic Add method(Algorithm 7), as there is a direct reachable path between them. After that we update the tc-list of all the vertices which are reachable to u and u's tc-list as well. For that we invoke *updateTCAfterAddE ()*(Algorithm 12) in the Line 167. This method first updates the tc-list of u which is given from Line 113-117, adds all the descendant of v to the descendant of u, by invoking the *addVTCtoU()*(in Line 121) method(Algorithm 9). Then backtrack the vertices(using backward DFS)which are reachable to u and then update all their descendants. We invoke *ADFSBW ()* method((Algorithm 10, in Line 122) to update all descendants. An example depicts in the Fig. 4, two thread T_1 and T_2 trying to add edge $(2, 5)$ and $(3, 6)$ to the TC-graph Fig.3 respectively. The corresponding update TC-graph is shown in the Fig.5.

Algorithm 1 It takes the input `key` and creates a new `Enode`

procedure *elist_t* CREATEE (*key* ↓)
  elist_t temp ← new elist_t;
  temp.val ← key;
  temp.lock ← NULL;
  temp.marked ←false;
  temp.enext ← NULL;
  return temp;
end procedure

Algorithm 2 It takes the input `key` and creates a new `TCnode`

procedure *tclist_t* CREATETC (*key* ↓)
  tclist_t temp ← new *tclist_t*;
  temp.val ← key;
  temp.lock ← NULL;
  temp.marked ← false;
  temp.tcnext ← NULL;
  return temp;
end procedure

Algorithm 3 It takes the input `key` and creates a new `Vnode`

```
procedure vlist_t CREATEV (key ↓)
  elist_t EHead ← createE(-infinity);
  elist_t ETail ← createE(+infinity);
  EHead.enext ← ETail;
  tclist_t TCHead ← createTC(-infinity);
  tclist_t TCTail ← createTC(+infinity);
  tclist_t newtc ← createTC(key);
  newtc.tcnext ← TCTail;
  TCHead.tcnext ← newtc;
  vlist_t temp ← new vlist_t;
  temp.val ← key;
  temp.lock ← NULL;
  temp.marked ← false;
  temp.enext ← EHead;
  temp.tcnext ← TCHead;
  temp.vnext ← NULL;
  return temp;
end procedure
```

Algorithm 4 It takes the input `pred` & `curr` of type generic $< T >$, where $< T >$ is either a vlist_t node, elist_t node or tclist_t node. It returns true with the invariant an unmarkable node is reachable from `Head`, else returns false.

```
procedure bool VALIDATE (< T > pred ↓, < T > curr ↓ )
  return (pred.marked = false ∧ curr.marked = false ∧ pred.next = curr);
end procedure
```

Algorithm 5 It takes input as vertex `key` and list head, returns the current location of key if key is present and returns true, else returns false . This method is wait-free.

```
procedure bool FIND (< T > head ↓, < T > curr ↑, key ↓,)
  pred ← head; curr ← pred.next;
  while (curr.val < key ) do
    pred ← curr; curr ← curr.next;
  end while
  if (curr.val = key) then
    return true;
  else
    return false;
  end if
end procedure
```

Algorithm 6 It takes input as `key` and returns the exact location of key in the list.

```
procedure void LOCATE (< T > head ↓, < T > pred ↑, < T > curr ↑, key ↓,)
  pred ← head; curr ← pred.next;
  while (curr.val < key ) do
    pred ← curr; curr ← curr.next;
  end while
  pred.lock(); curr.lock();
  if (Validate (pred↓, curr↓)) then // Algorithm 4
    return;
  else
    curr.unlock(); pred.unlock();
    return;
  end if
end procedure
```

Algorithm 7 It takes input as `key` and adds a node to the TC-graph if not present earlier, for successful addition it returns true, else returns false.

```
procedure bool ADD (< T > head ↓, key ↓,)
  Locate (< T > head ↓, < T > pred ↑, < T > curr ↑, key ↓);// Algorithm 6
  if (curr.val ≠ key)) then
    newt ← Create (key);// creates a new node of type < T >
    newt.next ← curr; // logical addition
    pred.next ← newt; // physical addition
    curr.unlock(); pred.unlock();
    return true;
  else
    curr.unlock(); pred.unlock();
    return false;
  end if
end procedure
```

Algorithm 8 It takes input as `key` and deletes a node from the TC-graph if present, for successful deletion it returns true, else returns false.

```
procedure bool DELETE (< T > head ↓, key ↓ ,)
  Locate (< T > head ↓, < T > pred ↑, < T > curr ↑, key ↓); // Algorithm 6
  if (curr.val = key)) then
    curr.marked ← true; // logical deletion
    pred.next ← curr.next; // physical deletion
    curr.unlock(); pred.unlock();
    return true;
  else
    curr.unlock();pred.unlock();
    return false;
  end if
end procedure
```

A new vertex `newv` is added by invoking *AddVertex* method(Algorithm 17). Each time this method is called with new vertex id, which is generated from the last vertex id plus one. This increment is done by atomic operation fetch-and-add (FAA). We assume all vertices have unique id and the system has unbounded number of such keys, once it is added to TC-Graph, will never assign this id to any other vertex. Each time a new TC node also created with that vertex id, say `newtc`. After that `newtc` is added to `newv` and then `newv` is inserted at the proper position of the vertex-list of TC-graph and it never affects the properties of TC-graph. The structure of `newv` is shown in the Fig 2.

5.2 Decremental Algorithms(SMDTC)

Like AddEdge, we say that an algorithm is decremental if it allows only deletion, it may be only edges or only vertices or both. For simplicity to prove the correctness we allow only deletion of edges in the paper. After deleting an edge, it may be difficult to restore the TC properties, as reachable path may get affected. After deleting an edge successfully, how quickly we restore the TC-graph instate of starting everything from the scratch is not an easy task. For that we proposed a decremental algorithm to maintain the transitive closure, we called it as SMDTC algorithm and the details of the algorithm is given bellow. We used backward depth-first search(BDFS) to find all reachable vertices to restore the affected TC after deleting an edge iff it violates the TC-graph.

To delete an edge, we invoke the $DeleteEdge(u, v)$ method presented in the Algorithm 16. First it checks the presence of vertices u and v by invoking the *Find ()* method (Algorithm 5) from Line 174 to 182. If any one of these vertices is not present or the edge is not present, we simply return false. After successful check of u and v, in the Line 182 we try to delete the edge node v(outgoing edge) in the u's edge-list and the edge node -u(incoming edge) in the v's edge-list. After successful deletion of both the edges, we delete `TCnode` v from the u's tc-list (Line 185) by invoking the generic Delete method(Algorithm 8). After that we update the tc-list of all the vertices which are reachable to v and u's tc-list as well. For that we invoke *updateTCAfterDelE ()*(Algorithm 14) in the Line 187. This method first updates the tc-list of u which is given from Line 139-143, deletes all the

Algorithm 9 It takes inputs, u & v of type vlist_t, adds all reachable vertices of v to `tc-list` of u.

procedure ADDVTCTOU($u \downarrow, v \downarrow$)
  tempu ← u.tcnext; tempv ← v.tcnext.tcnext;
  while (tempv.tcnext ≠ NULL)) **do**
    Add (tempu↓, tempv.val↓); // Algorithm 7
    tempv ← tempv.tcnext;
  end while
end procedure

Algorithm 10 It takes < T > elist_t `slHead` and vlist_t v. Do the backward DFS traversal. It processes the incoming edges.

procedure ADFSBW ($slHead \downarrow, v \downarrow$)
  for it ← slHead.enext to it.next ≠ NULL **do**
    if (it.val < 0) **then**// checks for incoming edges
      flag ← Find ($head \downarrow, curr \uparrow, (-1)it.val \downarrow,$)); //Algorithm 5
      if (flag = true) **then**
        addVTCtoU($curr \downarrow, v \downarrow$)
        ADFSBW ($curr.enext \downarrow, v \downarrow$); //Algorithm 10
      end if
    end if
  end for
end procedure

Algorithm 11 It takes inputs, u & v of type vlist_t v and removes all reachable vertices in v from tc-list of u.

```
procedure delVTCfromU(u ↓, v ↓)
  tempu ← u.tcnext; tempv ← v.tcnext.tcnext;
  while (tempv.tcnext ≠ NULL)) do
    Delete (tempu↓, tempv.val↓); // Algorithm 8
    tempv ← tempv.tcnext;
  end while
end procedure
```

Algorithm 12 Iterate all affected vertices and restore the TC-graph.

```
procedure bool updateTCAfterAddE (curr1 ↓, curr2 ↓)
  tempv ← curr1.tcnext.tcnext;
  while (tempv.tcnext ≠ NULL)) do
    Add (curr1.tcnext↓, tempv.val↓); // Algorithm 7
    tempv ← tempv.tcnext;
  end while
  for it ← curr1.enext to it.enext ≠ NULL do
    if (it.val < 0) then
      if (Find (n1 ↑, −it.val ↓)) then //Algorithm 5
        addVTCtoU(n1 ↓, curr2 ↓); //Algorithm 9
        ADFSBW (n1.enext ↓, curr2 ↓);// Algorithm 10
      end if
    end if
  end for
end procedure
```

Algorithm 13 It takes elist_t `slHead` and vlist_t v. Do the backward DFS traversal. Processed all incoming edges.

```
procedure DDFSBW (slHead ↓, v ↓)
  for it ← slHead.enext to it.next ≠ NULL do
    if (it.val < 0) then// checks for incoming edges
      flag ← Find (head ↓, curr ↑, −it.val ↓,)); //Algorithm 5
      if (flag = true) then
        delVTCfromU(curr ↓, v ↓)
        DDFSBW (curr.enext ↓, v ↓); //Algorithm 13
      end if
    end if
  end for
end procedure
```

Algorithm 14 Iterate all affected vertices and restore the TC-graph.

```
procedure bool updateTCAfterDelE (curr1 ↓, curr2 ↓)
  tempv ← curr1.tcnext.tcnext;
  while (tempv.tcnext ≠ NULL)) do
    Delete (curr1.tcnext↓, tempv.val↓); // Algorithm 8
    tempv ← tempv.tcnext;
  end while
  for it ← curr1.enext to it.enext ≠ NULL do
    if (it.val < 0) then
      if (Find (n1 ↑, −it.val ↓)) then //Algorithm 5
        delVTCfromU(n1 ↓, curr2 ↓); //Algorithm 11
        DDFSBW (n1.enext ↓, curr2 ↓);// Algorithm 13
      end if
    end if
  end for
end procedure
```

Algorithm 15 Adds both incoming and outgoing edges to the edge-list of vertex key_1,if it is not present earlier and then update all affected vertices of TC-graph. Otherwise returns false.

```
procedure bool AddEdge (key1 ↓,key2 ↓)
  flag1 ← Find (curr1 ↑, key1 ↓); //Algorithm 5
  flag2 ← Find (curr2 ↑, key2 ↓); //Algorithm 5
  if ( flag1 = false ∨ flag2 = false) then
    return false;
  end if
  flag1 ← Find (curr1 ↑, key1 ↓); //Algorithm 5
  if (flag1 = false) then
    return false;
  end if
  flag ← Add (curr1.enext, key2) ∧ Add (curr2.enext, −key1); //Algorithm 7
  if (flag = true) then
    flag4 ← Add (curr1.tcnext, key2); //Algorithm 7
    if (flag4 = true) then
      return updateTCAfterAddE (curr1 ↓, curr2 ↓); // Algorithm 12
    else
      return false;
    end if
  end if
end procedure
```

Algorithm 16 Deletes both incoming and outgoing edges from the edge-list of vertex key_1, if it is present earlier and then update all affected vertices of TC-graph. Otherwise returns false.

```
procedure bool DeleteEdge (key1 ↓,key2 ↓)
  flag1 ← Find (curr1 ↑, key1 ↓); //Algorithm 5
  flag2 ← Find (curr2 ↑, key2 ↓); //Algorithm 5
  if ( flag1 = false ∨ flag2 = false) then
    return false;
  end if
  flag1 ← Find (curr1 ↑, key1 ↓); //Algorithm 5
  if (flag1 = false) then
    return false;
  end if
  flag ← Delete (curr1.enext, key2) ∧ Delete (curr2.enext, (−1) ∗ key1); //Algorithm 8
  if (flag = true) then
    flag4 ← Delete (curr1.tcnext, key2); //Algorithm 8
    if ( thenflag4 = true)
      return updateTCAfterDelE (curr1 ↓ ,curr2 ↓); // Algorithm 14
    else
      return false;
    end if
  end if
end procedure
```

5.3 SMTC Algorithms

As we said earlier that dynamic algorithm are categorically classify in to three types depending on types of updates operations are allowed. A dynamic graph is said to be fully-dynamic if the update operations are restricted insertions and deletions of edges or vertices [4]. We said a graph algorithm is incremental if only insertions are allowed on the other hand a graph algorithm is decremental if only deletions are allowed. In the previous two subsections we discussed our proposed incremental(SMITC) and decremental(SMDTC) algorithms. For SMITC algorithm we allowed edge and vertices insertion and for SMDTC we allowed only edge deletion. To make our algorithm to be fully dynamic(SMTC) we allowed both edge insertion and deletion with fixed set of vertices.

Although SMTC algorithm works for all operations edge/vertices insertion or deletions. For simplicity as of now we allow only edge insertion or deletion. The main objective of SMTC algorithm is how quickly we restore the TC-graph after any updates. As we are using linked-list based set to represent edge-list, vertex-list and tc-list, it is easy to apply fine-grain synchronization to update the TC-graph. We use optimized searching to achieve better synchronization, the boolean `marked` field for each node helps that. We design wait-free searching algorithm, *Find ()*Algorithm 5, which searches a key present or not in the TC-graph without holding any locks. To make *Find ()* algorithm to be perfect wait-free, our current system has bounded number of bits which is enough for *Find ()* method to terminate the execution. Hence is it wait-free with respect to the bounded number of key generated by the system.

Algorithm 17 Adds a new vertex to the TC-graph, if not present earlier, else return false.

```
procedure bool AddVertex (key ↓)
  return Add (VHead ↓, key ↓); // Algorithm 7
end procedure
```

Algorithm 18 Checks key_1 has descendant wit key_2 or not.

procedure *bool* CHECKDESCENDANT ($key_1 \downarrow, key_2 \downarrow$)
  flag1 $\leftarrow$ Find ($Head \downarrow, curr1 \uparrow, key_1 \downarrow$); //Algorithm 5
  flag2 $\leftarrow$ Find ($Head \downarrow, curr2 \uparrow, key_2 \downarrow$); //Algorithm 5
  if ($flag1 = false \vee flag2 = false$) **then**
    return false;
  end if
  return Find ($curr1.tcnext \downarrow, curr \uparrow, key_2 \downarrow$); //Algorithm 5
end procedure

Algorithm 19 Computes the number of nodes which are reachable from a vertex at the given instance.

procedure *int* COUNTDESCENDANTS ($key \downarrow$)
  flag $\leftarrow$ Find ($Head \downarrow, curr \uparrow, key \downarrow$); //Algorithm 5
  if ($flag1 = false$) **then**
    return false;
  end if
  temp $\leftarrow$ curr.tcnext.tcnext;
  int count $\leftarrow$ 0;
  while temp.tcnext $\neq$ NULL **do**
    flag $\leftarrow$ Find ($Head \downarrow, curr \uparrow, key \downarrow$); //Algorithm 5
    if (flag = true $\wedge$ curr.marked = false) **then**
      count $\leftarrow$ count +1;
    end if
    temp $\leftarrow$ temp.tcnext;
  end while
  return count;
end procedure

5.4 Count and Check Descendant

Now we proposed an application of SMTC algorithm for descendant counting problem [3] on the on-line graph. To estimate or count number of descendant for a given vertex at the instance. So *checkDescendant (u, v)*, takes two arguments, u & v, checks whether v is descendant of u, means whether node v is present in the tc-list of u at the given instance. If it is, then there is a reachable path from the from vertex u to v. This algorithm is wait-free and running time be the tc-list size. Our TC-graph helps to achieve this without traversing the whole graph. Or starting everything from scratch after any dynamic updates.
Similarly, the *countDescendants (u)* method returns number of descendants currently present at node u, means we can count total number of vertices reachable from the vertex u. This algorithm helps to find the shortest path from a vertex to other vertex in a dynamic graph. Like checkDescendant, the countDescendants () method also wait-free. Above two algorithms have huge application in the area of strongly connected component(SCC), graphics, VLSI design, shortest path algorithm, bi-connected components, community detection, communication networks, and assembly planning etc. The checkDescendant and countDescendants methods are shown in the Algorithm 18 and 19 respectively.

5.5 Memory management

Our proposed algorithm depends on a explicit garbage collector(GC) for better memory management. We defined a separate GC method which is invoked by an independent thread in regular intervals of time. Our GC method is similar to Michael's Hazard Pointers technique [15] although it was designed for lock-free objects and we reclaim all three types of node. This GC thread does not affect the execution time.

6 The Correctness Proof

We now describe how our proposed algorithm SMTC is correct. A proof is is based on Timnat. et al.'s, [19]. We think the detail proof is very much important for concurrent data-structure and algorithms as without that, it is very hard to understand the races. Any directed graph is represented as TC-graph and it is collection of three types of lists. First, the *vertex-list*, each node of vertex-list has a Ehead & TChead to hold the adjacency edge-list and tc-list respectively. Secondly, *edge-list*, each vertex has a edge-list and finally, *tc-list*, each vertex has its reachable vertices. The TC-graph is interfaced with node id or key value `val`, boolean `marked` field and `tcnext` field. At any instance of time a node is considered to be part of TC-graph, if it is unmarked.

Proof Methodology We define the abstract TC-graph which always holds two invariant. Once the invariant holds for a node, it remain true. The first invariant is that, the node(tc or vertex or edge) can only physically change by pointer(`tcnext` or `vnext` or `enext`) and the key value of the node never change after initialization. Second, once a node is marked, it remain to be marked and it's next pointer never change until GC. For proving the correctness we use the four stages of any node similar like Timnat. et.al.'s, [19].

1. *Logical remove*: changing the `marked` filed false to true.
2. *Physical remove*: delinking the node from the list.
3. *Logical insertion*: Connecting new node's pointer to the node list.
4. *Physical Insertion*: making new logical node to a physical node, i.e. actual insertion.

We prove our algorithm using mathematical induction.

Lemma 1. *The history H generated by the interleaving of any of the methods of the TC-graph is linearizable.*

Lemma 2. *The methods* AddVertex*,* AddEdge *and* DeleteEdge *are deadlock-free.*

Proof Sketch: We prove all the $AddVertex$, $AddEdge$ and $DeleteEdge$ methods are deadlock-free by direct argument based of the acquiring lock on both the current and predecessor nodes.

1. $AddVertex$: the $AddVertex(key)$ method is deadlock-free because a thread always acquires lock on the *vnode* with smaller keys first. Which means, if a thread say T_1 acquired a lock on a $vnode(key)$, it never tries to acquire a lock on a *vnode* with key smaller than or equal to $vnode(key)$. This is true because the $AddVertex$ method acquires lock on the predecessor *vnode* from the $locateV$ method.
2. $AddEdge$: the $AddEdge(key_1, hey_2)$ method is deadlock-free because a thread always acquires lock on the *enode* with smaller keys first. Which means, if a thread say T_1 acquired a lock on a $enode(key_2)$, it never tries to acquire a lock on a *enode* of the vertex $vnode(key_1)$ with key smaller than or equal to $enode(key_2)$. This is true because the $AddEdge$ method acquires lock on the predecessor edge nodes of the vertex $vnode(key_1)$ from the $locateE$ method.
3. $DeleteEdge$: the $DeleteEdge(key_1, key_2)$ method is also deadlock-free, similar argument as $AddEdge$.

□

Lemma 3. *The methods countDescendants and checkDescendant are wait-free.*

Proof. The $countDescendants(key)$ method scans the tc-list of the vertex *vnode(key)*(if present) of the TC-graph starting from the $TCHead$, ignoring whether *tcnode*s are marked or not. It returns an integer value `count` depending on number of verteices are reachanble from the $vlode(key)$. If the *tcnode* and corresponding *vnode* are unmarked, it increments the local `count`. On the other hand, it never returns *zero* even if no other vertices are reachable, that case it return *one*, as each vertex is reachable itself. This *countDescendants* method is wait-free, because there are only a finite number of vertex keys one being searched for. By the observation of the code the vertices which are reachble must presnt in the $tc-list$ and which is finite, which says the traversal of *countDescendants* method is wait-free.

Similarly, the $checkDescendant(key_1, key_2)$ method first scans the vertex-list of the graph starting from the $VertexHead$, ignoring whether vertex nodes are marked or not. After successful check of vertex key_1 & key_2, it traverse the tc-list of $vnode(key_1)$. It returns a boolean $flag$ either *true* or $false$ depending on $tcnode(key_2)$ greater than or equal to the sought-after key in the tc-list of the vertex $vnode(key_1)$. If the desired *tcnode* is unmarked, it simply returns *true* and this is correct because the vertex-list is sorted as well as the tc-list of the vertex $vnode(key_1)$ is also sorted. On the other hand it returns $false$ if either $vnode(key_1)$ or $vnode(key_2)$ is not present or has been marked in the vertex-list or $tcnode(key_2)$ is not present or has been marked in the tc-list of the vertex *vnode*(key_1). This *checkDescendant* method is wait-free, because there are only a finite number of vertex keys that are smaller than the one being searched for as well as a finite number of tcnode keys that are smaller than the one being searched for in tc-list of any vertex. By observation of the code, *tcnode*(key_2) of the vertex *vnode*(key_1) is reachable from $TCHead$ even if vertex nodes or transitive closure node of *vnode*(key_1) are logically removed from the vertex-list. Therefore, each time the *Find* moves to a new transitive closure node, whose key value is larger key than the previous one. This can happen only finitely many times, which says the traversal of *checkDescendant* method is wait-free.

6.1 Linearization Points

In this section we identify the linearization point(LP) of our proposed methods. Before identifying the LP, we first consider Find, as it is used by most of the methods. It returns true if key value is present along with pair of pointers(`pred` & `curr`). For successful and unsuccessful *Find* the LP be Line 40 the last read of `curr.next`. The LP of a successful *AddEdge* is last node added to the tc-list in the Line 67, when a new transitive closure node is added, it is the physical insertion `pred.next` ← `newt`. For unsuccessful *AddEdge*, the LP is inside *Find* method where either of the vertex is not found or the edge node is present in the Line 40. The LP of a successful *DeleteEdge* is last node deleted from the tc-list, it is the logical deletion `curr.marked` ← `true` at the Line 78. For unsuccessful *AddEdge*, the LP is inside *Find* method where either of the vertex is not found or the edge node is not present in the Line 40.

7 Performance Analysis

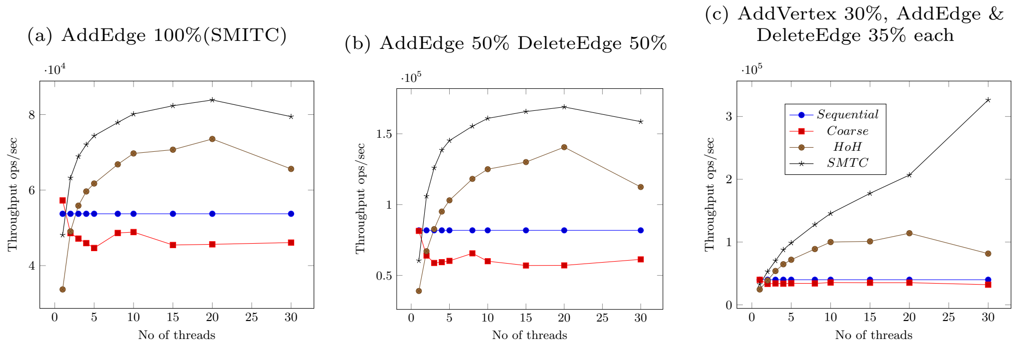

Fig. 6: TC-graph results-1

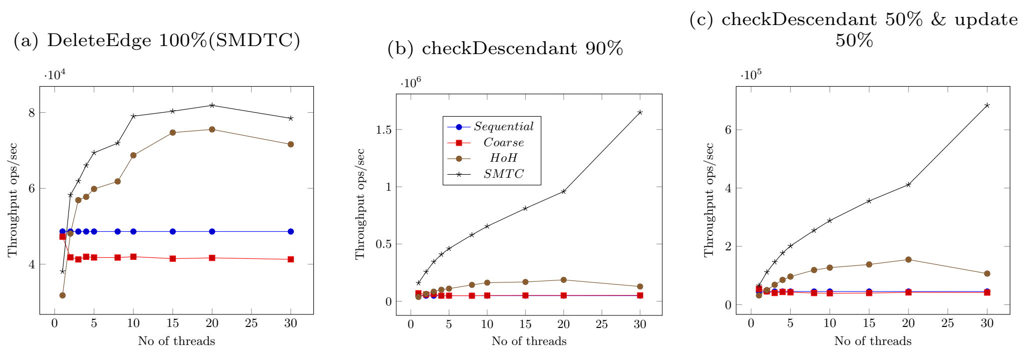

Fig. 7: TC-graph result-2 and descendant counting problem results.

In this section, we evaluate the performance of our SMTC algorithm. The source code available at https://github.com/Mukti0123/SMTC. It contains both fully and partial dynamic transitive

closure and some applications, such as descendant counting problem. We compare throughput with sequential, coarse-grain and hoh-locking.The methods are evaluated on a dual-socket, 10 cores per socket, Intel Xeon (R) CPU E5-2630 v4 running at 2.20 GHz frequency. Each core supports 2 hardware threads. Every core's L1 has 64k, L2 has 256k cache memory are private to that core; L3 cache (25MB) is shared across all cores of a processors. All the codes are compiled using the GCC C/C++ compiler (version 5.4.0) with -O3 optimization and Posix threads execution model.
Workload & methodology: We ran each experiment for 5 seconds, and measured the overall number of operations executed by all the threads(starting from 1,2,3,4,5,8,10,15,20 and 30). The graphs shown in the Fig 6 & 7 are the total number of operations executed by all threads. In all the tests, we ran each evaluation 5 times and took the average.
The algorithms we compare are, (1). Sequential(only one thread and no lock) (2). Coarse-grained(only one spin lock) (3). hoh-locking(fine-grained lock), (4), proposed SMTC algorithm. Each thread performed, in the Fig 6a, AddEdge 100%, in the Fig 6b, 50% AddEdge and 50% DeleteEdge, in the Fig 6c, 30% AddVertex and AddEdge 35% & DeleteEdge 35%. The Fig 6 shows the throughputs of *Sequential*, *Coarse-grain*, *hoh-locking* and *SMTC*. Similarly, the Fig 7 show, in Fig 7a AddEdge 100%, in the Fig 7b checkDescendant 90%, in Fig. 7c checkDescendant 50% and update 50%. After executing all above micro benchmarks, SMTC perform efficiently over sequential, coarse-grain and hoh-locking. The Fig 6 and 7 shows the throughput is increased between 3 to 6x with increasing number of thread depending on different workload distributions and applications.

8 Conclusion & Future Direction

In this paper, we present an on-line full dynamic algorithm for maintaining transitive closure of a directed graph in a shared memory architecture. The edges are added or deleted and vertices are added concurrently by fixed number of threads. To the the best of our knowledge, this is the first work to propose using linearizable concurrent directed graph. We have constructed TC-graph using three type of nodes, `Enode`, `Vnode` and `TCnode` which were build using list-based set. We provide an empirical comparison against sequential, course-grained and hoh-locking, with different workload distributions. We provide an empirical comparison against sequential, course-grained and hoh-locking. The results show our algorithm perform well an increasing number of threads. The throughput is increased between 3-6$\times$ depending on different workload distributions and applications. We believe that there are huge applications in the on-line graph. Finally, we show how the algorithm can be extended to descendant counting problem in on-line graph.

Currently the proposed update algorithms are blocking and deadlock-free. In the future, we plan to explore non-blocking(lock-free & wait-free) variant of all the methods of TC-graph. We believe that one can develop a better optimization techniques to handle the TC-graph restoring after the edges/vertices are added or deleted. Also we plan for other real world social graph applications.

References

1. David A. Bader, Jonathan W. Berry, Daniel Chavarría-Miranda, Kamesh Madduri, and Steven C. Poulos. Stinger: Spatio-temporal interaction networks and graphs (sting) extensible representation. 2009.
2. Michael A. Bender, Jeremy T. Fineman, Seth Gilbert, and Robert E. Tarjan. A New Approach to Incremental Cycle Detection and Related Problems. *ACM Trans. Algorithms*, 12(2):14, 2016. URL: http://doi.acm.org/10.1145/2756553, doi:10.1145/2756553.
3. Edith Cohen. Size-estimation framework with applications to transitive closure and reachability. *J. Comput. Syst. Sci.*, 55(3):441–453, December 1997. URL: http://dx.doi.org/10.1006/jcss.1997.1534, doi:10.1006/jcss.1997.1534.
4. Camil Demetrescu, David Eppstein, Zvi Galil, and Giuseppe F. Italiano. Algorithms and theory of computation handbook. chapter Dynamic Graph Algorithms, pages 9–9. Chapman & Hall/CRC, 2010. URL: http://dl.acm.org/citation.cfm?id=1882757.1882766.
5. Camil Demetrescu and Giuseppe F. Italiano. Trade-offs for fully dynamic transitive closure on dags: breaking through the o(n^2 barrier. *J. ACM*, 52(2):147–156, 2005. URL: http://doi.acm.org/10.1145/1059513.1059514, doi:10.1145/1059513.1059514.
6. Daniele Frigioni, Tobias Miller, Umberto Nanni, and Christos Zaroliagis. An experimental study of dynamic algorithms for transitive closure. *J. Exp. Algorithmics*, 6, December 2001. URL: http://doi.acm.org/10.1145/945394.945403, doi:10.1145/945394.945403.

7. Oded Green and David A. Bader. custinger: Supporting dynamic graph algorithms for gpus. *2016 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–6, 2016.
8. M. R. Henzinger and V. King. Fully dynamic biconnectivity and transitive closure. In *Proceedings of the 36th Annual Symposium on Foundations of Computer Science*, FOCS '95, pages 664–, Washington, DC, USA, 1995. IEEE Computer Society. URL: http://dl.acm.org/citation.cfm?id=795662.796250.
9. Monika R. Henzinger and Valerie King. Randomized fully dynamic graph algorithms with polylogarithmic time per operation. *J. ACM*, 46(4):502–516, July 1999. URL: http://doi.acm.org/10.1145/320211.320215, doi:10.1145/320211.320215.
10. Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming.* Morgan Kaufmann, 2008.
11. Maurice Herlihy and Nir 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. URL: http://dx.doi.org/10.1007/978-3-642-25873-2_22, doi:10.1007/978-3-642-25873-2_22.
12. Maurice P. Herlihy and Jeannette M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990. doi:http://doi.acm.org/10.1145/78969.78972.
13. Valerie King and Garry Sagert. A fully dynamic algorithm for maintaining the transitive closure. In *Proceedings of the Thirty-first Annual ACM Symposium on Theory of Computing*, STOC '99, pages 492–498, New York, NY, USA, 1999. ACM. URL: http://doi.acm.org/10.1145/301250.301380, doi:10.1145/301250.301380.
14. Chih-Chung Lin and Ruei-Chuan Chang. On the dynamic shortest path problem. *J. Inf. Process.*, 13(4):470–476, April 1991. URL: http://dl.acm.org/citation.cfm?id=105582.105591.
15. Maged M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Trans. Parallel Distrib. Syst.*, 15(6):491–504, June 2004. URL: http://dx.doi.org/10.1109/TPDS.2004.8, doi:10.1109/TPDS.2004.8.
16. Liam Roditty and Uri Zwick. Improved dynamic reachability algorithms for directed graphs. In *Proceedings of the 43rd Symposium on Foundations of Computer Science*, FOCS '02, pages 679–, Washington, DC, USA, 2002. IEEE Computer Society. URL: http://dl.acm.org/citation.cfm?id=645413.652176.
17. Liam Roditty and Uri Zwick. A fully dynamic reachability algorithm for directed graphs with an almost linear update time. In *Proceedings of the Thirty-sixth Annual ACM Symposium on Theory of Computing*, STOC '04, pages 184–191, New York, NY, USA, 2004. ACM. URL: http://doi.acm.org/10.1145/1007352.1007387, doi:10.1145/1007352.1007387.
18. Mikkel Thorup. Near-optimal fully-dynamic graph connectivity. In *Proceedings of the Thirty-second Annual ACM Symposium on Theory of Computing*, STOC '00, pages 343–350, New York, NY, USA, 2000. ACM. URL: http://doi.acm.org/10.1145/335305.335345, doi:10.1145/335305.335345.
19. Shahar Timnat, Anastasia Braginsky, Alex Kogan, and Erez Petrank. Wait-free linked-lists. In *Principles of Distributed Systems, 16th International Conference, OPODIS 2012, Rome, Italy, December 18-20, 2012. Proceedings*, pages 330–344, 2012. URL: https://doi.org/10.1007/978-3-642-35476-2_23, doi:10.1007/978-3-642-35476-2_23.
20. Christian Wulff-Nilsen. Faster deterministic fully-dynamic graph connectivity. In *Proceedings of the Twenty-fourth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '13, pages 1757–1769, Philadelphia, PA, USA, 2013. Society for Industrial and Applied Mathematics. URL: http://dl.acm.org/citation.cfm?id=2627817.2627943.
21. A Gitter Z Bar Joseph and I Simon. Studying and modelling dynamic biological processes using time-series gene expression data. In *Nature Publishing Group, a division of Macmillan Publishers Limited. All Rights Reserved.*, pages 552–564, 2012. URL: https://www.ncbi.nlm.nih.gov/pubmed/22805708.