SoftwareX 7 (2018) 273–280

Contents lists available at ScienceDirect

SoftwareX

journal homepage: www.elsevier.com/locate/softx

Original software publication

Decremental dynamic algorithm to trace mutually connected clusters

Deokjae Lee [a], S. Hwang [b], S. Choi [c], B. Kahng [a,*]

[a] CCSS, CTP and Department of Physics and Astronomy, Seoul National University, Seoul 08826, Republic of Korea
[b] Institute for Theoretical Physics, University of Cologne, Cologne, 50937, Germany
[c] Michigan Center for Theoretical Physics, Randall Laboratory of Physics, University of Michigan, Ann Arbor, MI 48109, USA

ARTICLE INFO

Article history:
Received 25 August 2017
Accepted 13 August 2018

Keywords:
Interdependent networks
Mutually connected clusters
Percolation
Hybrid phase transition
Euler tour tree

ABSTRACT

The structure and dynamics of interdependent networks model catastrophic failures in complex systems that are interdependent. Percolation transitions on these networks exhibit hybrid phase transitions, which have significant practical implications for the early detection of large-scale failures. While the computer simulation of the percolation transitions and related dynamics can effectively be reduced to the computation of mutually connected clusters, such a computation is nontrivial, and several algorithms to handle the task have been proposed. Here we introduce a C++ implementation of one of the algorithms. This implementation uses intrusive data structures and thus provides a greater flexibility for applications in which efficient memory access is critical. The data structures, which we provide as a part of the library, are also useful for general percolation problems.

Code metadata

Current code version	v1.0
Permanent link to code/repository used for this code version	https://github.com/ElsevierSoftwareX/SOFTX-D-17-00066
Legal Code License	MIT
Code versioning system used	git
Software code languages, tools, and services used	C++
Compilation requirements, operating environments & dependencies	Any OS with a compiler supporting C++11 standard. A compile time dependency on Boost.Optional (http://www.boost.org/).
If available Link to developer documentation/manual	
Support email for questions	lee.deokjae@gmail.com

1. Introduction

Catastrophic failures stemming from heavy mutual dependencies between complex systems are critical problems in modern society. For example, the mutual dependency between the power grid and the computer network caused a large abrupt blackout in Italy in 2003 [1]. Statistical physics of interdependent networks has been considered the primary paradigm to understand such phenomena since its recent introduction [1–5]. An important aspect of such catastrophic failures is that a small failure in a system can propagate to a large fraction of the system with a low but not negligible probability. More precisely, the size distribution of the total failures in a cascade follows a power law with an outlier whose size is a finite fraction of the entire system [2,6,7]. Percolation transitions in interdependent networks and the associated avalanche dynamics capture the essence of this phenomenon.

A percolation transition in interdependent networks is a hybrid phase transition (HPT) [2,6]. This has significant theoretical and practical implications. In HPTs, the order parameter exhibits a discontinuity at the transition point, whereas the system also shows critical phenomena in the vicinity of the transition point, manifested by divergent physical quantities. Thus, they possess the characteristics of both the first- and second-order phase transitions. HPTs have only been noticed recently by researchers and thorough theoretical understanding of HPTs is an ongoing and active area of research. In particular, for some practical applications,

* Corresponding author.
E-mail address: bkahng@snu.ac.kr (B. Kahng).

https://doi.org/10.1016/j.softx.2018.08.002

the divergent quantities may be used as precursors of catastrophic failures [8,9,2]. Studying such quantities therefore gives access to information that allows one to predict or perhaps even prevent such failures, making it a significant attribute of the paradigm.

Using computer simulation to study the percolation and avalanche dynamics of interdependent networks is a challenging task. It requires a dynamic (online) algorithm to trace the size distribution of the mutually connected clusters, as defined in the following section, while nodes or edges are being removed from the networks. Brute-force approaches based on previous algorithms used for ordinary percolation problems are not efficient enough for this process. Therefore, several algorithms have been proposed to specifically tackle this problem [10–12]. In this paper, we introduce an implementation of the algorithm developed in [11].

Besides its main purpose of dealing with interdependent networks, this implementation has some noteworthy properties that are useful for general percolation problems. First of all, it provides several efficient data structures, including an implementation of a fully dynamic algorithm for graph connectivity (HDT algorithm) developed in [13]. Here, the term connectivity denotes information about the connected components of a graph, such as the existence of a path between any pair of nodes. For the incremental percolation processes during which nodes or edges are added gradually, the disjoint-set forest is a simple and very efficient solution for simulations [14,15]. However, for the decremental percolation processes during which nodes or edges are removed gradually, an efficient solution is not trivial. The HDT algorithm is a recent advance in the computer science discipline, and is directly applicable to various decremental percolation processes.

Second, the implementations of the data structures are intrusive [16]. Intrusive data structures are widely used in the development of performance- and resource-critical software such as operating systems and games [17,18]. An intrusive implementation of a data structure does not concern the allocation and release of the memory to store the data. The users must allocate and release the memory explicitly and provide hooks to be used by the implementation (see Section 4.4). Whereas this choice imposes more boilerplates in general, the explicit control of the memory comes with the flexibility and opportunity to optimize memory access. Being able to experiment and control the data cache miss rate is crucial for improving runtime performance in current computer architectures.

2. Percolation and avalanche process in interdependent networks

Let us consider two networks $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$, where V_1 and V_2 are the sets of nodes in each network, and E_1 and E_2 are the sets of edges in each network. The mutual dependencies between these networks are dictated by 'special edges' that link a node of one network to a node of the other. We will refer to them as 'interdependency edges,' and to the two networks as 'interdependent networks' [1]. Here, both the edges in a graph and the interdependency edges between graphs are assumed to be undirected.

To describe the interdependency structure, it is convenient to introduce the set of nodes interdependent on a node n as $I(n)$, i.e., every node $m \in I(n)$ is linked to n by an interdependency edge. Furthermore, by abuse of notation, we define $I(W) = \bigcup_{n\in W} I(n)$ for any subset W of either V_1 or V_2.

A mutually connected cluster (MCC) M is a pair (H_1, H_2) of subgraphs H_is of G_is for $i = 1, 2$ with the following conditions, given the subgraphs $H_i = (W_i, F_i)$:

1. Each pair of nodes in W_i is connected by at least one path in H_i.
2. $I(W_i) \neq \emptyset$ and $I(W_1) = W_2$ and $I(W_2) = I(W_1)$.
3. M is maximal in the sense that any addition of nodes or edges to H_i violates 1 or 2.

We refer to H_1 and H_2 as the projection of M on G_1 and G_2 respectively. Each forms a connected component in the respective network. If $|W_1| + |W_2| = O(|V_1| + |V_2|)$, M is called the giant mutually connected cluster (GMCC). As a special case, if each node has one and only one interdependency edge, each network is partitioned by the set of all projections, and the projections of any MCC to the two networks have the same size (number of nodes). Otherwise, some nodes may not belong to any MCC, and projections of one MCC to the two networks may have different sizes.

To model the propagation of failures in mutually dependent complex systems, let us assume that a node is functional if and only if (a) it belongs to the giant cluster of its network, and (b) each of its interdependent node also belongs to the giant cluster of the respective network. This criterion of functionality defines a connected component of functional nodes in each network, and it coincides with the projection of the GMCC [1,19].[1]

We can model the failure of a node by deleting or separating it from the GMCC. By the nature of the GMCC, all interdependent neighbors of a failed node will simultaneously fail, causing some nodes to be separated from the largest cluster of the network. These separated nodes will cause additional failures, which in turn trigger more failures, and so on. The process iterates until no more nodes fail. This propagation of failures is called an avalanche. The algorithm we present here is essentially a formalization of this avalanche process for MCCs.

With the use of our fast implementation, one can efficiently study percolation problems by tracing the size of the GMCC and controlling some aspects of the networks, such as the mean degree, number of initially failed nodes, degree distributions, and so on [1,19,21,22,12]. Various properties of the avalanche dynamics, for example the mean size and the duration of the propagation, are also quantities of interest [1,2,23,12,6].

3. Algorithm to trace MCCs dynamically

In Algorithm 1, we present a high-level description of the algorithm that traces the MCCs while nodes or edges are being deleted one by one. The description is slightly more general than the description in [11], in the sense that in our current setting we allow each node to have zero or multiple interdependencies. To start, the algorithm requires every network to have only one connected component. If not, we add auxiliary edges to make each network connected, and they will later be removed by applying the algorithm. The algorithm is essentially a process checking the following two assertions when edges are deleted: (a) If two nodes have no path in a network, then they cannot be in the same MCC. (b) If a node has two interdependent nodes that are in different connected components, then the node cannot belong to any MCC. Hence, an efficient way to maintain the information on the connectivity during the process is crucial for ensuring the performance of the algorithm.

For this purpose, the algorithm maintains a spanning forest for each network. In contrast to the common usage of spanning forests in which they represent the connected components of the networks, we employ them here to store the projections of MCCs. By construction, initially there is only one MCC, and each spanning

[1] One may assume that each node must have at least one interdependent node that belongs to the giant cluster instead of assuming all interdependent nodes belong to the giant cluster [20]. This leads to a definition of MCC different from ours.

forest has only one connected component. Calling the function DELETE-EDGE with an edge e updates the spanning forests so that the spanning forests represent projected MCCs in the new configuration. Subsequent calls of DELETE-EDGE update the spanning forests in an efficient way and the size distribution of the MCCs can be obtained from them without a substantial cost at any time. This can readily be extended to implement node removal. Since deleting a node is equivalent to deleting all of its edges, the function DELETE-NODE can be implemented by replacing line 4 in Algorithm 1 with a line that enqueues all edges of the given node.

The data structure for the spanning forest should provide the following operations efficiently:

(i) Query on the size of each connected component (spanning tree). This is the main quantity of interest in percolation problems; the size of an avalanche is also obtained from this.
(ii) Query on the existence of a path between two arbitrary nodes. Lines 42 and 47 in Algorithm 1 can be implemented with this.
(iii) Delete an edge included in the forest and find another edge that will replace the deleted one if the connectivity is not represented faithfully by the spanning forest after the deletion. This is used at lines 19 and 20 in Algorithm 1.

The spanning forest represented by a collection of the Euler tour trees provides the first two operations with amortized $O(\log N)$ time complexity [24]. As a better alternative, the HDT algorithm provides a data structure for the spanning forest supporting all these operations with amortized $O(\log^2 N)$ time complexity, by managing multiple layers of Euler tour trees for each spanning tree [13]. Although the HDT algorithm guarantees a good amortized time complexity of (iii), in some cases, the naive Euler tour trees complemented with a brute-force implementation of (iii), which simply searches for a replacement edge by a random sweeping, could be faster [11]. To be specific, in our experiments on interdependent Erdös–Rényi random networks, the first candidate edge scanned by the brute-force implementation is actually a replacement edge with a high probability. Thus the additional cost to maintain multiple layers of Euler tour trees in the HDT algorithm is wasted in this case. Nevertheless, to provide the greatest flexibility to users, we present both implementations of the spanning forest, i.e., the HDT algorithm, and the Euler tour trees with the brute-force approach for (iii).

Algorithm 1 High-level description of the algorithm.

```
 1: function DELETE-EDGE((v, w): edge)
 2:     p ← an empty queue    ▷ nodes which cannot belong to any MCC
 3:     q ← an empty queue    ▷ edges to be removed from the spanning forest
 4:     enqueue (v, w) into q
 5:     while p or q is not empty do
 6:         if q is not empty then
 7:             (x, y) ← dequeue from q
 8:             INACTIVATE((x, y), p, q)
 9:         end if
10:         if p is not empty then
11:             z ← dequeue from p
12:             ALIENATE(z, p, q)
13:         end if
14:     end while
15: end function
16:
17: function INACTIVATE((v, w): edge, p: queue, q: queue)
18:     if (v, w) is in the spanning forest then
19:         remove (v, w) from the spanning forest
20:         (x, y) ← FIND-REPLACEMENT-EDGE(v, w)    ▷ See Section 3.
21:         if the edge (x, y) is found then
22:             add (x, y) to the spanning forest
23:         else
24:             CHECK-INCONSISTENCY(v, w, p, q)
25:         end if
26:     end if
27: end function
28:
29: function ALIENATE(x: node, p: queue, q: queue)
30:     ▷ Here an alienated node denotes a node that does not belong to any MCC.
31:     if x is not marked as alienated then
32:         enqueue all edges of x into q
33:         enqueue all nodes interdependent on x into p
34:         mark x as alienated
35:     end if
36: end function
37: function CHECK-INCONSISTENCY(v: node, w: node, p: queue, q: queue)
38:     c1 ← smaller one between the connected component of v and w
39:     c2 ← larger one between the connected component of v and w
40:     for all x ∈ nodes in c1 do
41:         for all y ∈ interdependent nodes of x do
42:             if y is interdependent on a node in c2 then
43:                 ▷ y cannot belong to any MCC.
44:                 enqueue y into p
45:             end if
46:             for all (y, z) ∈ edges of y do
47:                 if z is interdependent on a node in c2 then
48:                     ▷ y and z cannot be in the same MCC.
49:                     enqueue (y, z) into q
50:                 end if
51:             end for
52:         end for
53:     end for
54: end function
```

4. Implementation

Our implementation is provided as a C++ header-only library. It contains four general data structures, as well as the implementation of the algorithm described in Algorithm 1. It uses features of the C++11 standard. For some compilers you need to set a flag to specify the standard. It also uses a third-party library Boost.Optional [25]. If the compiler supports the C++17 standard, `std::optional` can drop-in replace `boost::optional` completely.

4.1. Implementation of Algorithm 1

The file `DecrementalMCC.hpp` provides the algorithm using the Euler tour trees and the brute-force approach for the operation (iii) in Section 3. The alternative implementation using the HDT algorithm is provided in `DecrementalMCCWithHDT.hpp`. They share the same public interface, but are defined in different namespaces. The former is in the namespace `Snu::Cnrc::DecrementalMCC`, and the latter in `Snu::Cnrc::DecrementalMCCWithHDT`.

The nodes and edges are represented by the classes `DecrementalMCCNode` and `DecrementalMCCEdge` respectively. If you want to equip additional data on the nodes or edges, you can write your own data types, say `Node` and `Edge`, by extending `DecrementalMCCNodeMixin<Node, Edge>` and `DecrementalMCCEdgeMixin<Node, Edge>` respectively. The inheritance carries no virtual functions and thus no runtime cost. The trait class `DecrementalMCCTrait<Node, Edge>`, which defines constraints about the node and edge types, works under the hood to take care of the intrusive structure. You can specialize the

trait template for Node and Edge instead of extending the mixin classes, but usually this is tedious and has no advantage.

Once the choice of spanning forest algorithm is made, we are ready to set up and modify the interdependent networks. The operations to add an edge between two nodes and query about the projected MCCs are provided as static methods of DecrementalMCCAlgorithm<Node, Edge>::SpanningForest (See Listing 1 and Section 4.2). Similarly, DecrementalMCCAlgorithm<Node, Edge> provides static methods to set interdependencies between nodes in different networks and obtain MCCs after a deletion of an edge: interconnect to set interconnections, initialize to obtain the initial MCCs, and disconnect to obtain MCCs after the deletion of an edge. At this point, it is worth noting that we do not allow users to modify the interdependency structure or add edges during the process as our algorithm is for processes involving only deletions of edges. Hence, the methods connect and interconnect should not be used once initialize is called.

At any time point of the process, various network properties can be retrieved using the following methods. However, to correctly use them, some understanding of implementation detail is required. The implementation picks out a representative node for each projected MCC. Accordingly, there is one set for each network, and each set is represented by the class DecrementalMCCAlgorithm<Node, Edge>::ClusterRepresentativeSet. This class has identical interface with std::set<Node*>; its elements are representative nodes, sorted in decreasing order by the size of the connected component that they represent. You can query about a projected MCC by giving its representative node to the methods of DecrementalMCCAlgorithm<Node, Edge>::SpanningForest.

Similarly, the class DecrementalMCCAlgorithm<Node, Edge>::ClusterSizeDistribution, which has identical interface with std::map<unsigned int, unsigned int>, is used to represent the size distribution of the projected MCCs. The method disconnect takes ClusterRepresentativeSet and ClusterSizeDistribution representing current state as arguments and updates them upon the removal of an edge. Sometimes we are only interested in the size distribution of the MCCs and not in the detailed configuration of the MCCs. If each node has one and only one interdependency, the size distribution of the MCCs projected on a network already contains full information without tracing the set of the representatives. Thus, in this case we can save runtime resources by not tracing the set of representatives. In this consideration, we provide multiple overloads of disconnect method; one for the size distribution in a network, another one for the two sets of representatives, and the last one for the size distribution and the set of representatives in a network. See Listing 1 for details.

4.2. Spanning forest implementations

The file EulerTourTreeSpanningForest.hpp implements a data structure for spanning forests based on the Euler tour trees and the brute-force approach to the operation (iii) in Section 3. The file HDTSpanningForest.hpp implements a spanning forest data structure based on the HDT algorithm. The data structures are separately defined in namespaces Snu::Cnrc::EulerTourTreeSpanningForest and Snu::Cnrc::HDTSpanningForest respectively.

Our intrusive implementation follows the pattern already described in Section 3. For the HDT-based implementation, we provide template classes HDTSpanningForestNode, HDTSpanningForestEdge, HDTSpanningForestNodeMixin<Node, Edge>, HDTSpanningForestEdgeMixin<Node, Edge>, and HDTSpanningForestTrait<Node, Edge>. For the implementation based on Euler tour trees, we similarly provide EulerTourTreeSpanningForestNode, EulerTourTreeSpanningForestEdge, and so on.

The operations are defined as static methods of the template classes EulerTourTreeSpanningForestAlgorithm<Node, Edge> and HDTSpanningForestAlgorithm<Node, Edge>. The three operations described in Section 3 are provided by the methods clusterSize, hasPath, and disconnect as the same interface for both implementations. STL-compatible interfaces to iterate over the nodes and edges in a connected component are also provided. See Listing 2 for details.

The node and edge mixin classes used with DecrementalMCCAlgorithm<Node, Edge> extend appropriate mixins for the spanning forest data structure. Thus they can be used with the spanning forest operations, for example, see line 15: or 46: in Listing 1.

4.3. Euler tour tree implementation

The Euler tour tree is a way to represent a tree as a sequence of nodes [24]. Operations on the sequences corresponding to the split or join of trees can be implemented easily. If we store the sequence in a self-balanced tree, deciding whether or not any given two nodes are in the same tree is also supported efficiently. We use the AVL tree, which is a widely used balanced binary tree [26], to store the sequence (see Section 4.4). The spanning forests described in 4.2 use the Euler tour trees to represent spanning trees. This is implemented in the file EulerTourTree.hpp. It defines template classes EulerTourTreeNode<Node, Edge>, EulerTourTreeEdge<Node, Edge>, EulerTourTreeTrait<Node, Edge> and so on in the namespace Snu::Cnrc::EulerTourTree. The interface follows a similar pattern with the spanning tree implementations. Adding an edge between two nodes that are in the same tree has an undefined consequence and therefore must be avoided. We do not demonstrate its use in a code listing because it is very similar to that of the spanning forests implementations.

4.4. AVL tree to represent a sequence of elements

The file AVLSequence.hpp implements an intrusive AVL tree to store a sequence of elements. For our purpose, any balanced binary tree can be used in place of AVL. While balanced binary trees are usually used as a container to store and search elements, we needed to represent a sequence of elements instead. The usual implementations of the C++ standard library contain an implementation of a balanced binary tree, but since they are not designed to store a sequence of elements, we had to implement one by ourselves.

In our implementation, the inorder traversal of the tree represents the sequence of the elements. The data structure provides the split and join of sequences in $O(\log N)$ time in the worst case, where N is the size of the shorter sequence. Whether or not any two given elements are in the same tree can be determined also in $O(\log N)$ worst-case time complexity.

The header file defines a namespace Snu::Cnrc::AVLSequence, in which three template classes AVLSequenceNodeTrait, AVLSequenceNodeMixin, and AVLSequenceAlgorithm are defined.

Our implementation of the AVL tree is also intrusive, similar to our other data structures. This means the data type to be stored in the tree must provide an interface to store and read the height of the subtree, as well as pointers to the parent, left child, and right child.

Listing 1 Example code listing for the use of DecrementalMCCAlgorithm

```
namespace MCC = Snu::Cnrc::DecrementalMCC;
// or namespace MCC = Snu::Cnrc::DecrementalMCCWithHDT;

using Node = MCC::DecrementalMCCNode;
using Edge = MCC::DecrementalMCCEdge;
using Algo = MCC::DecrementalMCCAlgorithm<Node, Edge>;
using Forest = Algo::SpanningForest;

std::vector<Node> nodes1(3); // nodes in the network #1.
std::vector<Node> nodes2(3); // nodes in the network #2.
std::vector<Edge> edges1(3); // edges in the network #1.
std::vector<Edge> edges2(3); // edges in the network #2.

// Set edges.
Forest::connect(nodes1[0], nodes1[1], edges1[0]);
Forest::connect(nodes1[1], nodes1[2], edges1[1]);
Forest::connect(nodes1[2], nodes1[0], edges1[2]);
Forest::connect(nodes2[0], nodes2[1], edges2[0]);
Forest::connect(nodes2[1], nodes2[2], edges2[1]);

// Set interconnections.
Algo::interconnect(nodes1[0], nodes2[0]);
Algo::interconnect(nodes1[1], nodes2[1]);
Algo::interconnect(nodes1[1], nodes2[2]);
Algo::interconnect(nodes1[2], nodes2[2]);

// Compute the MCCs. Setting edges and interconnections must be done
// before this. `reprs1` and `reprs2` contain the representatives of MCCs
// projected on the network #1 and network #2 respectively.
Algo::ClusterRepresentativeSet reprs1, reprs2;
Algo::initialize(nodes1, nodes2, reprs1, reprs2);

// Other options: In the next two alternatives, `cdist1` contains the size
// distribution of MCCs projected on the network #1. It is not so useful if
// some nodes have zero or multiple interdependencies.
//
// Algo::ClusterSizeDist cdist1;
// Algo::initialize(nodes1, nodes2, cdist1);
//
// or
//
// Algo::initialize(nodes1, nodes2, cdist1, reprs1);

// Iterate over all MCCs
for(const Node* n : reprs1) {
  Forest::ConstCluster c1 = Forest::cluster(*n);
  Forest::ConstCluster c2 = Forest::constCluster(
    *Algo::interconnectedNeighbors(*n)[0]
  );
  // Iterate over the nodes in the MCC of `n` projected on the network #1.
  for(const Node& m : c1) {
    // Iterate over the edges of `m` whose the other end is also in the MCC.
    for(const Edge& e : Forest::edges(m)) {
      // Do something with `m` and `e`.
    }
  }
  // Iterate over the nodes in the MCC of `n` projected on the network #2.
  for(const Node& m : c2) {
    // Iterate over the edges of `m` whose the other end is also in the MCC.
    for(const Edge& e : Forest::edges(m)) {
      // Do something with `m` and `e`.
    }
  }
}

// Print the projection size distribution.
 for(auto i : cdist1) {
   std::cout << i.first << "\t" << i.second << std::endl;
 }

// Disconnect edges2[1].
// Now only one MCC consisting of nodes1[0] and nodes2[0] remains.
Algo::disconnect(edges2[1], reprs1, reprs2);

// Call an appropriate alternative instead of the above,
// according to the version of `initialize` you have called.
//
```

```
78: // Algo::disconnect(edges2[1], cdist1);
79: // Algo::disconnect(edges2[1], cdist1, reprs1);
80:
81: assert(*reprs1.begin() == &nodes1[0]);
82: assert(*reprs2.begin() == &nodes2[0]);
83: // assert(cdist1.size() == 1 && cdist1[1] == 1);
84:
85: // Disconnect edges1[0]. The MCC profile does not change.
86: Algo::disconnect(edges1[0], reprs1, reprs2);
```

Listing 2 Example code listing for spanning forests

```
1: namespace NS = Snu::Cnrc::HDTSpanningForest;
2: // or namespace NS = Snu::Cnrc::EulerTourTreeSpanningForest;
3:
4: struct Node;
5: struct Edge;
6:
7: struct Node : public NS::HDTSpanningForestNodeMixin<Node, Edge> {
8:   Node(int n) : name(n) {}
9:   bool operator==(const Node& n) const { return this == &n; }
10:  bool operator!=(const Node& n) const { return this != &n; }
11:  int name;
12: };
13:
14: struct Edge : public NS::HDTSpanningForestEdgeMixin<Node, Edge> {};
15:
16: using Forest = NS::HDTSpanningForestAlgorithm<Node, Edge>;
17:
18: Node n0(0), n1(1), n2(2), n3(3);
19: Edge e0, e1, e2;
20:
21: // Set edges.
22: Forest::connect(n0, n1, e0);
23: Forest::connect(n1, n2, e1);
24: Forest::connect(n2, n0, e2);
25:
26: // Methods to query about the connected component of a node.
27: assert(Forest::clusterSize(n0) == 3);
28: assert(Forest::cluster(n0).size() == 3);
29: assert(Forest::constCluster(n0).size() == 3);
30: assert(Forest::findClusterRep(n0) == Forest::findClusterRep(n1));
31: assert(Forest::findClusterRep(n0) != Forest::findClusterRep(n3));
32: assert(Forest::hasPath(n0, n1));
33: assert(! Forest::hasPath(n0, n3));
34:
35: // Print node names in the connected component of `n0`.
36: // `cluster` has overloads for `const Node&` and `Node&`.
37: // You can also use `constCluster`.
38: for(Node& n : Forest::cluster(n0)) {
39:   std::cout << n.name << std::endl;
40: }
41:
42: // Print edges of `n0`.
43: // You can also use `edges(Node&)` and `edges(const Node&)`.
44: for(const Edge& e : Forest::constEdges(n0)) {
45:   std::cout << Forest::node1(e).name << "-"
46:             << Forest::node2(e).name << std::endl;
47: }
48:
49: // `disconnect` returns true if the cluster is split.
50: assert(! Forest::disconnect(e0));
51: assert(Forest::disconnect(e2));
52: assert(! Forest::hasPath(n0, n1));
53: assert(Forest::clusterSize(n0) == 1);
```

The interface is defined in the template class `template<class T>AVLSequenceNodeTrait`. The template parameter `T` is the data type to be stored in the tree and the user needs to provide a specialization of the trait template for `T`. However, the typical implementation of the interface is trivial and thus a default implementation is provided. Moreover, if the data type `T` extends `AVLSequenceNodeMixin <T>`, no template specialization is needed and the default implementation of `AVLSequenceNodeTrait<T>` is sufficient. The operations on the tree are defined as static methods of the class `AVLSequenceAlgorithm<T>`. Listing 3 provides an example code demonstrating these methods.

5. Summary

We implemented an algorithm to trace MCCs dynamically in the process where nodes or edges are being deleted. The algorithm

Listing 3 Example code listing using AVLSequenceAlgorithm

```
using namespace Snu::Cnrc::AVLSequence;

struct Data : public AVLSequenceNodeMixin<Data> {
  Data(int value) : value(value) {}
  int value;
};

using A = AVLSequenceAlgorithm<Data>;

// Initially each element forms a sequence of each own.
Data d0(0), d1(1), d2(2), d3(3), d4(4);

// Now there are two sequences [0, 1, 2], [3, 4].
A::insertNodeAfter(d0, d1);
A::insertNodeAfter(d1, d2);
A::insertNodeBefore(d4, d3);

// True
&A::findRoot(d0) == &A::findRoot(d1);

// False
&A::findRoot(d0) == &A::findRoot(d3);

// Join the two sequences.
A::join(d2, d3);

// Split. Now we have [0, 1], [2, 3, 4];
A::splitBefore(d2);

// Join again. This is equivalent to A::join(d1, d2);
A::join(A::findTail(d0), A::findHead(d3));

// Print the sequence
for(Data& d : A::containerView(A::findRoot(d0)))
    std::cout << d.value << std::endl;

// Print the sequence. A::previous and A::next return
// boost::optional<std::reference_wrapper<Data>>.
for(auto d = A::previous(d1); d = A::next(*d); )
    std::cout << d->get().value << std::endl;
```

is based on four data structures, which also prove to be useful in the study of general percolation problems. The entire library containing the algorithm as well as the data structures is provided as a C++ header-only library. We demonstrated the usage of the implementation and the data structures.

Acknowledgments

This work was supported by the National Research Foundation of Korea by grant no. NRF-2014R1A3A2069005 (BK) and NRF-2017R1A6A3A11033971 (DL).

References

[1] Buldyrev SV, Parshani R, Paul G, Stanley HE, Havlin S. Catastrophic cascade of failures in interdependent networks. Nature 2010;464(7291):1025–8. http://dx.doi.org/10.1038/nature08932.

[2] Baxter GJ, Dorogovtsev SN, Goltsev AV, Mendes JFF. Avalanche collapse of interdependent networks. Phys Rev Lett 2014;109(24):248701. http://dx.doi.org/10.1103/PhysRevLett.109.248701.

[3] Reis SDS, Hu Y, Babino A, Andrade Jr. JS, Canals S, Sigman M, et al. Avoiding catastrophic failure in correlated networks of networks. Nat Phys 2014;10(10):762–7. http://dx.doi.org/10.1038/nphys3081.

[4] Kivela M, Arenas A, Barthelemy M, Gleeson JP, Moreno Y, Porter MA. Multilayer networks. J Complex Netw 2014;2(3):203–71. http://dx.doi.org/10.1093/comnet/cnu016.

[5] Lee K-M, Min B, Goh K-I. Towards real-world complexity: an introduction to multiplex networks. Eur Phys J B 2015;88(2):48. http://dx.doi.org/10.1140/epjb/e2015-50742-1.

[6] Lee D, Choi S, Stippinger M, Kertész J, Kahng B. Hybrid phase transition into an absorbing state: percolation and avalanches. Phys Rev E 2016;93(4):042109. http://dx.doi.org/10.1103/PhysRevE.93.042109.

[7] Lee D, Choi W, Kertész J, Kahng B. Universal mechanism for hybrid percolation transitions. Sci Rep 2017;7:5723. http://dx.doi.org/10.1038/s41598-017-06182-3.

[8] Scheffer M, Bascompte J, Brock WA, Brovkin V, Carpenter SR, Dakos V, et al. Early-warning signals for critical transitions. Nature 2009;461(7260):53–9. http://dx.doi.org/10.1038/nature08227.

[9] Scheffer M, Carpenter SR, Lenton TM, Bascompte J, Brock W, Dakos V, et al. Anticipating critical Transitions. Science 2012;338(6105):344–8. http://dx.doi.org/10.1126/science.1225244.

[10] Schneider CM, Araújo NAM, Herrmann HJ. Algorithm to determine the percolation largest component in interconnected networks. Phys Rev E 2013;87(4):043302. http://dx.doi.org/10.1103/PhysRevE.87.043302.

[11] Hwang S, Choi S, Lee D, Kahng B. Efficient algorithm to compute mutually connected components in interdependent networks. Phys Rev E 2015;91(2):022814. http://dx.doi.org/10.1103/PhysRevE.91.022814.

[12] Grassberger P. Percolation transitions in the survival of interdependent agents on multiplex networks, catastrophic cascades, and solid-on-solid surface growth. Phys Rev E 2015;91(6):062806. http://dx.doi.org/10.1103/PhysRevE.91.062806.

[13] Holm J, de Lichtenberg K, Thorup M. Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. J ACM 2001;48(4):723–60. http://dx.doi.org/10.1145/502090.502095.

[14] Galler BA, Fisher MJ. An improved equivalence algorithm. Commun. ACM 1964;7(5):301–3. http://dx.doi.org/10.1145/364099.364331.

[15] Newman M, Ziff R. Fast monte Carlo algorithm for site or bond percolation. Phys Rev E 2001;64(1):016706. http://dx.doi.org/10.1103/PhysRevE.64.016706.

[16] Assmann U. Invasive software composition. Berlin, New York: Springer; 2003.

[17] Love R. Linux Kernel development. 3rd ed. Upper Saddle River, NJ: Developer's library, Addison-Wesley; 2010. oCLC: 844983182.

[18] Nystrom R. Game programming patterns. Genever Benning; 2014.

[19] Son S-W, Grassberger P, Paczuski M. Percolation transitions are not always sharpened by making networks interdependent. Phys Rev Lett 2011;107(19):195702. http://dx.doi.org/10.1103/PhysRevLett.107.195702.

[20] Shao J, Buldyrev SV, Havlin S, Stanley HE. Cascade of failures in coupled network systems with multiple support-dependence relations. Phys Rev E 2011;83(3):036116. http://dx.doi.org/10.1103/PhysRevE.83.036116.
[21] Kim JY, Goh K-I. Coevolution and correlated multiplexity in multiplex networks. Phys Rev Lett 2013;111(5):058702. http://dx.doi.org/10.1103/PhysRevLett.111.058702.
[22] Min B, Yi SD, Lee K-M, Goh K-I. Network robustness of multiplex networks with interlayer degree correlations. Phys Rev E 2014;89(4):042811. http://dx.doi.org/10.1103/PhysRevE.89.042811.
[23] Zhou D, Bashan A, Cohen R, Berezin Y, Shnerb N, Havlin S. Simultaneous first- and second-order percolation transitions in interdependent networks. Phys Rev E 2014;90(1):012803. http://dx.doi.org/10.1103/PhysRevE.90.012803.
[24] Henzinger MR, King V. Randomized fully dynamic graph algorithms with polylogarithmic time per operation. J ACM 1999;46(4):502–16. http://dx.doi.org/10.1145/320211.320215.
[25] [Boost C++ Libraries]. URL http://www.boost.org/.
[26] Cormen TH, editor. Introduction to algorithms. 3rd ed. Cambridge, Mass: MIT Press; 2009.