Working Title

M. Sanghavi, S. Tadepalli, M. Nakayama

January 2013

1 Algorithms

We now provide optimized algorithms for generating all possible trees and updating the Q matrix. A tree corresponds to a set of particular components failing, and cascading failures starting from different states can have the same set of components failing. Hence, a particular tree may correspond to several different transitions. Our algorithm generates each possible tree only once and determines all the transitions to which this tree corresponds. This avoids generating the same tree numerous times, as was originally done in [**?**]. Because the number of trees grows exponentially in the number of components in the model, our current algorithm significantly reduces the total computational effort. Moreover, rather than building each new tree from scratch, as was done in [**?**], our current algorithm builds larger trees from smaller ones already considered, leading to additional savings in the computation.

Computing the rate of a given tree depends on the state from which the cascading failure began and the set of components that fails in the cascade. Each component can possibly cause a set of other components that fails when it itself fails, however, it is not necessary that a component's entire set of components fails in each instance. While building a tree, each time a component fails, some subset of the components capable of being triggered to fail, actually fails. A supertree is built from a tree by filling in these missing components that were not actually triggered, but would have been triggered had the entire set of components failed in each instance. We do not actually build the supertree in our application, but instead build a breadth-first history to represent one. A breadth-first history optimally accounts

for the contribution of the components that could have failed, but did not, to the rate of the tree. We explain the process of building a breadth-first history and using it after to compute the rate of a tree in Algorithms 2 and 3, respectively.

Algorithm 1, SeedTrees, starts the tree generation and initializes the necessary data structures for Algorithm 2, AddTreeLevel. AddTreeLevel, adds a new level to an existing tree in a recursive fashion, updates the cumulative failed probability for the tree, as wells as builds out the tree's breadth-first history. Algorithm 3, ComputeTreeRate, computes the rate of a completed tree for all the transitions it corresponds to using the cumulative failed probability and the breadth-first history populated in AddTreeLevel. We will now discuss each algorithm in detail. Line numbers from each the algorithms are given in their corresponding texts within [ ].

1.1 Algorithm 1

Cascading failures per component are described in Γ, which is represented as an array of sets of component types indexed by component types, with each set containing the component types that can be caused to fail, if the index component type fails. We start by iterating through *compSet*, the set of components, to choose a root component, *rootC*, for an initial tree with one node. [1] We then initialize the following data structures: *level*, a dynamic array to hold all the failed nodes in a level in breadth-first order; *nFailed*, a list that counts the number of failed components of each type; and *BFHist*, a data structure that is the Breadth First History of a tree. *BFHist* is implemented as an array of linked lists indexed by component type. Each linked list stores the respective parents of nodes exclusive to the supertree (i.e. the nodes that did not fail but could have) and stores the symbol @ for the nodes that did fail in the tree of the type of the index which points to the linked list in *BFHist*. *BFHist* plays the role of a supertree in determining whether to include the complement probabilities of nodes that did not fail. A complement probability is included for a node that did not fail only iff there are still components of the type available in the system. The number of available components of a type are given by redundancy of the component type minus the number failed in the from state, minus the number failed in the tree thus far in breadth-first order. [$2-4$]

Since the root must fail, we initialize *level* with *rootC*. We used the symbol @ to denote when a component has failed. We add the symbol @ *BFHist* at the index of type *rootC*. We update *nFailed* as well by setting the counter for type *rootC* to 1. $[5-7]$

If the component at the root, *rootC*, cannot cause any other components to fail, only the trivial tree of one node can be made. We then evaluate this single-node tree's rate in ComputeTreeRate because it cannot be grown further. Otherwise, for *rootC* with a nonempty Γ, i.e. can cause other types of components to fail, we call AddTreeLevel to proceed with building taller trees by adding another level to the current tree. $[8-13]$

In treeRate we only keep track of the cumulative component-affected probabilities $\phi_{i,j}$ for the transition rate of the tree, which entails multiplying $\phi_{i,j}$ for all edges from parent i to child j in the tree. We multiply the $1-\phi$ terms and the $n * \lambda$ later in ComputeTreeRate as they depend on the start state of the failure transition. There are no failed nodes that have been triggered in the tree thus far, so we pass 1 as the value for treeRate to the subroutines AddTreeLevel and ComputeTreeRate, which are given in Algorithms 2 and 3, respectively.

1.2 Algorithm 2

In Algorithm 2 AddTreeLevel, given a tree, we determine all the possibilities for the next level that can be added to the current tree by taking the Cartesian product of the power sets of Γ for each of the failed components. For each configuration of the next level we recursively call AddTreeLevel and get a different tree. We use $\mathcal{P}$ to denote the power set-like operation on an ordered set. In our case, each subset of the power set maintains the relative ordering in the original set.

We implement this power set operation by generating all possible binary numbers with $|\sum_{i=1}^{|level|} \Gamma_{level[i]}|$ bits. A total of $|\prod_{i=1}^{|level|} 2^{\Gamma_{level[i]}}|$ such binary numbers are generated. In the binary number, 1 denotes a failed node, 0 denotes a node that could have failed but did not fail. If it so happens that we have a tree where none of the leaf nodes can cause any other components to fail

Algorithm 1 SeedTrees(Γ)

where Γ is an array of ordered sets that describes which components can cause which other components to fail

1: **for** *rootC* $\in$ *compSet* **do**
2: *level* = [ ]; {dynamic array of failed components at tree's current level}
3: *nFailed* = $(0, 0, \ldots, 0)$; {counts failed components of each type}
4: *BFHist* = $(( ), ( ), \ldots, ( ))$; {an array of linked lists that keeps a breadth-first history of trees, array is indexed by component type, linked list for each component type stores parents in breadth-first order}
5: add *rootC* to *level*;
6: *nFailed*[*rootC*] = 1;
7: add @ to *BFHist*[*rootC*]; {signifies one component of type rootC has failed}
8: **if** Empty(Γ_{rootC}) **then**
9: ComputeTreeRate(*nFailed*, *BFHist*, *treeRate*, *rootC*);
10: **else**
11: AddTreeLevel(*level*, *nFailed*, *BFHist*, 1, *rootC*);
12: **end if**
13: **end for**

(i.e., have empty Γs), we get no nextLevelPossibilities. [1] If there are no nextLevelPossibilities we immediately proceed to ComputeTreeRate. [2−4]

Otherwise, we choose one possible choice for the failed components in the next level to work with from the nextLevelPossibilities. [5] To find out whether any new children will be added in the upcoming next level, we create a boolean *addedChildFlag*; initially with the value of False. [6] For each node *parentC* in the current level that acts as a parent, potentially causing other nodes to fail, we iterate through all of its possible children, i.e., through its Γ. If any of these children actually fail, then they will be members of the set *oneNextLevelPossibility*. [7 − 9] Now if the redundancy of the component type we just added as a child, has already been exhausted, it cannot fail and hence our tree is invalid and we move on to the next possibility. [10 − 12]

If it is indeed possible to add a child of the type of *childC*, we flag this occurrence and update the corresponding data structures. We add 1 to *nFailed* at the index of type *childC*. We add the symbol @ to *BFHist* at the index of type *childC* to mark that a failure has occurred at this location in the tree.We update tree rate with the component-affected probability of parent triggering the child to fail. [13 − 16] If the child does not actually fail but could have failed (because there is still operational components of this type at this point), then we add *parentC* to *BFHist*, at the index of the type of the child. *BFHist* comes in use later in ComputeTreeRate, to determine when to multiply the current tree rate with the $1 - \phi_{i,j}$ terms for the components that did not fail but could have. [17 − 19]

We do the above updates to data structures for each potential parent node in *level* and each of its children in *oneNextLevelPossibility*. [7−21] If at least one child has been added, we add another level to the current tree. [22−23] Otherwise, this tree has not changed in this pass through AddTreeLevel and we call ComputeTreeRate to compute the rate of the finalized tree. [24−26] We make sure that trees are not double counted because once a tree passes through AddTreeLevel unmodified, it is processed and discarded. We do not make duplicate trees because each oneNextLevelPossibility is unique.

Algorithm 2 AddTreeLevel(*level*, *nFailed*, *BFHist*, *treeRate*, *rootC*)

where *level* describes failed components,
nFailed counts failed components by type,
BFHist is Breadth First History,
treeRate is a cumulative probability of comps that failed,
rootC is the root component of the current tree

1: $nextLevelPossibilities = \overset{|level|}{\underset{i=1}{\times}} \mathcal{P}(\Gamma_{level[i]})$;
{Builds set of all possible nodes in next level as Cartesian product of power sets of Γs}
2: **if** Empty(*nextLevelPossibilities*) **then**
3: ComputeTreeRate(*nFailed*, *BFHist*, *treeRate*, *rootC*);
{current tree cannot be grown further because its leaf nodes have empty Γ}
4: **end if**
5: **for** *oneNextLevelPossibility* ∈ *nextLevelPossibilities* **do**
6: *addedChildFlag* = False;
7: **for** *parentC* ∈ *level* **do**
8: **for** *childC* ∈ $\Gamma_{parentC}$ **do**
9: **if** *childC* ∈ *oneNextLevelPossibility* **then**
10: **if** *nFailed*[*childC*] == Redundancy(*childC*) **then**
11: goto line 3; {invalid tree, requires more comps than available in system}
12: **end if**
13: *addedChildFlag* = True;
14: *nFailed*[*childC*] = *nFailed*[*childC*] + 1;
15: add @ to *BFHist*[*childC*]; {signifies one component of type childC has failed}
16: *treeRate* = *treeRate* ∗ $\phi_{parentC,\, childC}$;
{update rate with ϕ}
17: **else**
18: add *parentC* to *BFHist*[*childC*]; {signifies one component of type childC has not failed, but was present in $\Gamma_{parentC}$}
19: **end if**
20: **end for**
21: **end for**
22: **if** *addedChildFlag* **then**
23: AddTreeLevel(*oneNextLevelPossibility*, *nFailed*, *BFHist*, *treeRate*, *rootC*);
{tree can be grown further}
24: **else**
25: ComputeTreeRate(*nFailed*, *BFHist*, *treeRate*, *rootC*);
{current tree is completed because it cannot be grown further}
26: **end if**
27: **end for**

1.3 Algorithm 3

In Algorithm 3, ComputeTreeRate, we update all failure transitions in the Q matrix where the tree is applicable by adding the tree rate to each. Each tree is mutually exclusive of the others so tree rates are combined by addition.

We use *'* to denote a state independent of an environment. *x'* is a particular from state (independent of environment) from the state space *S'* (also independent of environment.) [1] *prodNotFailedProb* denotes the product of the probabilities of all the nodes in the tree that did not fail, but could have. [2] Within each *x'*, for each component type we calculate the number of components by type that are available in the system as redundancy minus the number already failed in the from state. The number of available components, *compsAvailable*, determines until which point, while traversing through the *BFHist*, we can still have components of a certain type that could have failed but did not fail. [4] We can have no more components that *could* fail when they have been exhausted or equivalently, their total number that is failed equals redundancy. Each time we encounter the symbol @ in the *BFHist* we reduce the number available by one. $[5-7]$ If there are components still available we update *prodNotFailedProb*. $[8-9]$ As soon as *compsAvailable* reaches zero we break because that type has been exhausted and hence there are no more could have failed for the type. $[10-12]$

Once we have finished calculating the *prodNotFailedProb* we loop through all environments. [15] We generate to states y (with an environment e) by adding *nFailed* to *x'*. $[16-19]$ Invalid to states will be generated in some instances, because simply adding *nFailed* will cause component types' number failed to exceed their redundancy. $[20-22]$ The failure rate of the root as per the model is calculated. [23] With all parts of the rate calculation done, we update the failure transition's cell in the Q matrix with the tree rate. [25]

Algorithm 3 ComputeTreeRate(*nFailed*, *BFHist*, *treeRate*, *rootC*)

where *level* describes failed components,
nFailed counts failed components by type,
BFHist is Breadth First History,
treeRate is a cumulative probability of comps that failed,
rootC is the root component of the current tree

1: **for** $x' \in S'$ **do**
2: *prodNotFailedProb* = 1; {cumulative probability of comps that could have failed but did not}
3: **for** $comp \in compSet$ **do**
4: $compsAvailable = \text{Redundancy}(comp) - x[comp]$;
5: **for** $parentC \in BFHist[comp]$ **do**
6: **if** $parentC == @$ **then**
7: $compsAvailable = compsAvailable - 1$;
8: **else if** $compsAvailable > 0$ **then**
9: $prodNotFailedProb = prodNotFailedProb * (1 - \phi_{parentC,\, comp})$;
10: **else**
11: break; {compsAvailable must equal 0 so no more $1 - \phi$}
12: **end if**
13: **end for**
14: **end for**
15: **for** $e \in envSet$ **do**
16: Initialize y as a state with no components failed and environment e;
17: **for** $comp \in compSet$ **do**
18: $y[comp] = x[comp] + nFailed[comp]$;
19: **end for**
20: **if** y is not a valid state **then**
21: continue;
22: **end if**
23: $rootFailureRate = (\text{Redundancy}(rootC) - x'[rootC]) * \lambda_{rootC,\, e}$;
24: **end for**
25: $Q(x, y) = Q(x, y) + rootFailureRate * treeRate * prodNotFailedProb$;
26: **end for**