Working Title

M. Sanghavi, S. Tadepalli, M. Nakayama

January 2013

1 Algorithms

We now provide efficient 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 i can possibly cause any subset of components from a set Γ_i to simultaneously fail when i fails; i.e., not all components in Γ_i actually fail in a given cascade. We also consider the supertree from a given tree by adding in the components that did not fail from the Γ_i that were used, which is necessary to compute the rate of a tree. We do not actually build the supertree in our algorithm, but instead build a *breadth-first history* to keep track of the information necessary to compute a tree's rate. A breadth-first history accounts for the contribution to the tree's rate of the components that could have failed (i.e., belong to the set Γ_i of a node of type i that actually did fail), but did not. We explain the process of building a breadth-first history and using it to compute the rate of a tree in Algorithms 2 (AddTreeLevel) and 3 (ComputeTreeRate), respectively.

SeedTrees (Algorithm 1), starts the tree generation and initializes the necessary data structures for AddTreeLevel (Algorithm 2). AddTreeLevel adds a new level to an existing tree in a recursive fashion, updates the cumulative failed probability for the tree for the components that actually failed, as well as builds the tree's breadth-first history. ComputeTreeRate (Algorithm 3) 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 square brackets [ ].

1.1 SeedTrees

Cascading failures per component are described in Γ that is an array of sets of component types. Each Γ_i is the set of component types that can be caused to fail when a component type i 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 the current level in breadth-first order; *nFailed*, a list that counts the number of failed components of each type in the tree; 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 provided their component type's redundancy was not exhausted) and stores the symbol @ for the nodes that *did* fail in the tree. *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 if there are still components of the type available in the system at that point. The number of available components of a type is given by the 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 $[2-4]$. For a transition (x, y), we define the "from" state as x, and the "to" as y.

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

If the component *rootC* at the root cannot cause any other components to fail, only the trivial tree of one node can be made with this type of root. We then evaluate this single-node tree's rate in ComputeTreeRate because it cannot be grown further. If $\Gamma_{rootC} \neq \varnothing$, i.e., it can cause other types of components to fail, then 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 product of the 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 "from" state of the failure transition. There are no failed nodes that have been caused to fail in a cascade, so we pass 1 as the value for treeRate to the subroutines AddTreeLevel and ComputeTreeRate.

1.2 AddTreeLevel

In 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 at the bottom level. For each next level we recursively call AddTreeLevel. 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-like 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

Algorithm 1 SeedTrees(Γ)

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

```
for rootC ∈ compSet do
    level = [ ]; {dynamic array of failed components at tree's current level}
    nFailed = (0, 0, ..., 0); {counts failed components of each type}
    BFHist = (( ), ( ), ..., ( )); {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}
    add rootC to level;
    nFailed[rootC] = 1;
    add @ to BFHist[rootC]; {signifies one component of type rootC has failed}
    if Empty(Γ_rootC) then
        ComputeTreeRate(nFailed, BFHist, treeRate, rootC);
    else
        AddTreeLevel(level, nFailed, BFHist, 1, rootC);
    end if
end for
```

fail. If it so happens that we have a tree where none of the leaf nodes at the bottom level can cause any other components to fail (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 variable *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 tried to add 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 necessary 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 the parent causing 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.

1.3 ComputeTreeRate

In ComputeTreeRate, for a given tree, we determine all of the transitions (x, y) in the Q-matrix that use this tree, and then update the total rates of each of those transitions (x, y) by adding in the rate of the current tree to the current rate for (x, y). However, even if several transitions use the same tree, the rate of the tree may differ for those transitions, so we need to compute a separate tree rate for each transition. The failure rate of the root and cumulative product of complement probabilities are transition dependent, leading to different rates for the same tree shared by multiple transitions.

Let x' be a particular "from" state's failed component types count (which is independent of environment) from the state space S' (also independent of environment) [1]. The variable *prodNotFailedProb* denotes the product of the complement probabilities of all the nodes in the tree that did not fail, but could have provided their component type's redundancy had not been exhausted [2]. For each component type in 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 *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 cannot have components that *could have failed but did not* after a point in the tree where their type has been exhausted. A type is exhausted when the total number of failed components of the type in the system, i.e. the sum of the components failed in the "from" state and in the tree in breadth-first order until the current node, equals the redundancy of the type. 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 then 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 since the same tree can be used for transitions occurring in different environments. [15]. We generate "from" states by adding an environment e to x'. from We generate "to" states y (with an environment e) by adding *nFailed* to x' [16 − 20]. Invalid "to" states will be generated in some instances, because simply adding *nFailed* will cause component types' number failed to exceed their redundancy. [21 − 23]. The failure rate of the root is calculated as the number of components of the *rootC*'s type up in the system, multiplied by the failure rate of the component in the environment of y. [24]. With all parts of the rate calculation done, we update the failure transition's cell in the Q matrix with the tree rate [26].

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 components that failed,
rootC is the root component of the current tree

1: $nextLevelPossibilities = \bigtimes_{i=1}^{|level|} \mathcal{P}(\Gamma_{level[i]})$;
{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 at the bottom level have empty Γ}
4: **end if**
5: **for** $oneNextLevelPossibility \in nextLevelPossibilities$ **do**
6:   $addedChildFlag$ = False;
7:   **for** $parentC \in level$ **do**
8:     **for** $childC \in \Gamma_{parentC}$ **do**
9:       **if** $childC \in oneNextLevelPossibility$ **then**
10:         **if** $nFailed[childC]$ == Redundancy($childC$) **then**
11:           goto line 3; {invalid tree, requires more components 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 component -affected probability}
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**

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

where *level* describes failed components,
nFailed counts failed components of each type in the tree,
BFHist is breadth-first history,
treeRate is a cumulative probability of components that failed,
rootC is the root component of the current tree

for $x' \in S'$ **do**
  prodNotFailedProb = 1; {cumulative product of complement probabilities of components that could have failed but did not}
  for *comp* $\in$ *compSet* **do**
    compsAvailable = Redundancy[*comp*] − *x*[*comp*];
    for *parentC* $\in$ *BFHist*[*comp*] **do**
      if *parentC* == @ **then**
        compsAvailable = *compsAvailable* − 1;
      else if *compsAvailable* > 0 **then**
        prodNotFailedProb = *prodNotFailedProb* $*$ $(1 - \phi_{parentC,\, comp})$;
      else
        break; {compsAvailable must equal 0 so no more $1 - \phi$}
      end if
    end for
  end for
  for $e \in$ *envSet* **do**
    Initialize x as a state with x' of components failed and environment e;
    Initialize y as a state with no components failed and environment e;
    for *comp* $\in$ *compSet* **do**
      y[*comp*] = x[*comp*] + *nFailed*[*comp*];
    end for
    if y is not a valid state **then**
      continue;
    end if
    rootFailureRate = (Redundancy[*rootC*] − x[*rootC*]) $* \lambda_{rootC,\, e}$;
  end for
  $Q(x, y) = Q(x, y) +$ *rootFailureRate* $*$ *treeRate* $*$ *prodNotFailedProb*;
end for

References