Working Title

M. Sanghavi, S. Tadepalli, M. Nakayama

January 2013

1 Algorithms

We now provide highly optimized algorithms for generating all possible subTrees and updating the Q Matrix. Since the number of trees for a model is exponential, we generate each unique subTree and its corresponding superTree (also unique) only once. We iterate through the Q Matrix and update all transitions where the subTree is applicable. This is the tree based approach to filling in the Q Matrix, which is mentioned in the remark in Section 4.2 of the preceding paper. [**?**] Generating subtrees only once ensures the bottleneck of tree generation in DECaF takes the least possible execution time. The superTree is represented by keeping a Breadth First History of the subtree. We also generate larger subTrees by adding one level at at time to already generated subTrees. This ensures that work done to generate an existing tree can be reused. By not starting at the root each time we need to generate a tree saves DECaf a tremendous amount of execution time. We make optimizations to other areas in DECaF as well.

Algorithm 1 SeedSubTrees starts the tree generation and initializes the necessary data structures for the recursion in Algorithm 2 AddSubTreeLevel.

Γ describes cascading failures with details about which components can cause which other components to fail. We start by iterating through the model's component set, *compSet* to choose a root component, *rootC* for an initial tree with one node. 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 differentiated by type and *BFHist* a data structure that is the Breadth First History of a subTree. It is implemented as an array of linked lists indexed by component type that stores parents of both nodes, which did and did not fail in a subTree. *BFHist* plays the role of a superTree to calculate the complement rates of nodes that did not fail.

Since the root must fail, we add *rootC* to *level*. We use @ to denote when a component has failed. We add @ *BFHist* at the index of type *rootC*. We update *nFailed* as well by setting the counter for type *rootC* to 1.

If a component cannot cause any other components to fail only the trivial subTree of one node can be made. We then evaluate and process this single-node subTree's rate because it cannot be grown further. Otherwise for all types of components that have a nonempty Γ, or equivalently, can cause other types of components to fail, we call AddSubTreeLevel to proceed with building height-wise larger subTrees.

In subTreeRate we only keep track of the cumulative product of failureRates by multiplying $\phi i, j$ for all edges from parent i to child j in the subTree. There are no failed nodes that have been triggered in the tree thus far, so we pass 1 to AddSubTreeLevel and ComputeTreeRates.

In Algorithm 2 AddSubTreeLevel, given a subTree, we determine all the possibilities for subTrees with one more level by taking the Cartesian product of the power sets of Γ for each of the failed components. We use $\mathcal{P}$ to denote the power set like operation on a 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 in line 1 by generating all possible $|\sum_{i=1}^{|level|} \Gamma_{level[i]}|$ bit binary numbers. 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 or equivalently, have empty Γs, we get no nextLevelPossibilities. If there are no nextLevelPossibilities we immediately proceed to ComputeTreeRates.

Otherwise, we choose one possibility to work with. 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 or equivalently through its Γ. If any of these children actually fail then they will be members of the set *oneNextLevelPossibility*. 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 try the next possibility.

If it is indeed possible to add a child of the type of *childC* we flag this oc-

Algorithm 1 SeedSubTrees(Γ)

where Γ is an ordered set that describes which components can cause which other components to fail

for $rootC \in compSet$ **do**
  $level$ = [ ]; {dynamic array of failed components at subTree's current level}
  $nFailed = (0, 0, \ldots, 0)$; {counts failed components of each type}
  $BFHist = ((\,), (\,), \ldots, (\,))$; {an array of linked lists that keeps a breadth-first history of subTrees, 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**
    ComputeTreeRates($nFailed$, $BFHist$, $subTreeRate$, $rootC$);
  else
    AddSubTreeLevel($level$, $nFailed$, $BFHist$, 1, $rootC$);
  end if
end for

currence and update the corresponding data structures. We add 1 to *nFailed* at the index of type of ChildC. We add @ *BFHist* at the index of type *chidC* to mark that a failure has occurred at this location in the subTree. We update subTree rate with the failure rate of parent triggering the child to fail. If the child does not actually fail but could have failed we add its would have been parent to *BFHist* at the index of the type pf the child. This come in use later in ComputeTreeRates.

We do the above updates to data structures for each potential parent node in *level* and each of its children in *oneNextLevelPossibility*.

If at least one child has been added we add another level to the current tree, otherwise this tree has not changed in this pass through AddSubTreeLevel and we call ComputeTreeRates on it.

We make sure that trees are not double counted because each oneNextLevelPossibility is unique and each time a tree does not change, it is processed and discarded. This ensures that each time a tree comes in it either has an additional level or has been enumerated with another next level possibility.

In Algorithm 3 ComputeTreeRates we fill in all transitions in the Q matrix where the subtree is applicable.

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.) *prodNotFailedProb* denotes the product of the probabilities of all the nodes in the tree not failing. Within each *x'*, for each component type we calculate the number of components by type that are available in the system as redundancy minus already failed in the from state. The number of available components, *compsAvailable*, determines at which point, while traversing through the *BFHist*, we can no longer have any components of a certain type that could have failed but did not fail. We can have no more components that *could* fail when they have been exhausted or equivalently, their total number failed equals redundancy. Each time we encounter @ in the *BFHist* we reduce the number available by one. If there are components still available we update *prodNotFailedProb*. 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.

Once we have finished calculating the *prodNotFailedProb* we loop through all environments. We generate to states y (with an environment) by adding *nFailed* to *x'*. Invalid to states will be generated in some instances because simply adding *nFailed* will cause component types' number failed to exceed

Algorithm 2 AddSubTreeLevel(*level*, *nFailed*, *BFHist*, *subTreeRate*, *rootC*)

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

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: ComputeTreeRates(*nFailed*, *BFHist*, *subTreeRate*, *rootC*);
{current subTree cannot be grown further because its leaf nodes 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 subtree, 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: $subTreeRate = subTreeRate * \phi_{parentC,\, childC}$;
{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: AddSubTreeLevel(*oneNextLevelPossibility*, *nFailed*, *BFHist*, *subTreeRate*, *rootC*);
{subTree can be grown further}
24: **else**
25: ComputeTreeRates(*nFailed*, *BFHist*, *subTreeRate*, *rootC*);
{current subTree is completed because it cannot be grown further}
26: **end if**
27: **end for**

their redundancy. The failure rate of the root as per the model is calculated. With all parts of the rate calculation done, we update the failure transition. We add because each tree is independent of the others.

Algorithm 3 ComputeTreeRates(*nFailed*, *BFHist*, *subTreeRate*, *rootC*)

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

```
for x' ∈ S' do
  prodNotFailedProb = 1; {cumulative probability of comps that could have failed but did not}
  for comp ∈ compSet do
    compsAvailable = Redundancy(comp) − x[comp];
    for parentC ∈ BFHist[comp] do
      if parentC == @ then
        compsAvailable = compsAvailable − 1;
      else if compsAvailable > 0 then
        prodNotFailedProb = prodNotFailedProb ∗ (1 − φ_{parentC, comp});
      else
        break; {compsAvailable must equal 0 so no more 1 − φ}
      end if
    end for
  end for
  for e ∈ envSet do
    Initialize y as a state with no components failed and environment e;
    for comp ∈ 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]) ∗ λ_{rootC, e};
  end for
  Q(x, y) = Q(x, y) + rootFailureRate ∗ subTreeRate ∗ prodNotFailedProb;
end for
```