Title Title 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 larger subTrees.

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

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 Γ's}
2: **for** *oneNextLevelPossibility* $\in$ *nextLevelPossibilities* **do**
3:   *addedChildFlag* = False;
4:   **for** *parentC* $\in$ *level* **do**
5:     **for** *childC* $\in \Gamma_{parentC}$ **do**
6:       **if** *childC* $\in$ *oneNextLevelPossibility* **then**
7:         **if** *nFailed*[*childC*] == Redundancy(*childC*) **then**
8:           goto line 3; {invalid subtree, requires more comps than available in system}
9:         **end if**
10:         *addedChildFlag* = True;
11:         *nFailed*[*childC*] = *nFailed*[*childC*] + 1;
12:         add @ to *BFHist*[*childC*]; {signifies one component of type childC has failed}
13:         $subTreeRate = subTreeRate * \phi_{parentC,\,childC}$;
{update rate with ϕ}
14:       **else**
15:         add *parentC* to *BFHist*[*childC*]; {signifies one component of type childC has not failed, but was present in $\Gamma_{parentC}$}
16:       **end if**
17:     **end for**
18:   **end for**
19:   **if** *addedChildFlag* **then**
20:     AddSubTreeLevel(*oneNextLevelPossibility*, *nFailed*, *BFHist*, *subTreeRate*, *rootC*);
{subTree can be grown further}
21:   **else**
22:     ComputeTreeRates(*nFailed*, *BFHist*, *subTreeRate*, *rootC*);
{current subTree is completed because it cannot be grown further}
23:   **end if**
24: **end for**

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 $x' \in S'$ **do**
  $prodNotFailedProb = 1$; {cumulative probability of comps that could have failed but did not}
  for $comp \in compSet$ **do**
    $compsAvailable = \text{Redundancy}(comp) - x[comp]$;
    for $parentC \in BFHist[comp]$ **do**
      if $parentC == @$ **then**
        $compsAvailable = compsAvailable - 1$;
      else if $compsAvailable > 0$ **then**
        $prodNotFailedProb = prodNotFailedProb * (1 - \phi_{parentC,\, comp})$;
      end if
    end for
  end for
  for $e \in envSet$ **do**
    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
    $rootFailureRate = (\text{Redundancy}(rootC) - x[rootC]) * \lambda_{rootC,\, e}$;
  end for
  $Q(x, y) = Q(x, y) + rootFailureRate * subTreeRate * prodNotFailedProb$;
end for