Working Title

M. Sanghavi, S. Tadepalli, M. Nakayama

March 2013

1 Model

We work with the stochastic model of [2], which considers the evolution over time of a repairable dependability system operating in a randomly changing environment. The system consists of a collection $\Omega = \{1, 2, \ldots, N\}$ of $N < \infty$ component types. Each component type $i \in \Omega$ has a redundancy $1 \leqslant r_i < \infty$, and the r_i components of type i are assumed to be identical. A component can be either operational (up) or failed (down).

The environment changes randomly within a set $\mathcal{E} = \{0, 1, 2, \ldots, L\}$. For example, we can think of the environment as representing the current load on the system, and if there are two possible environments, 0 and 1, then 0 (resp., 1) may represent a low (resp., high) load. Once the environment enters $e \in \mathcal{E}$, it remains there for an exponentially distributed amount of time with rate $\nu_e > 0$, after which the environment changes to e' with probability $\delta_{e,e'} \geqslant 0$, where $\delta_{e,e} = 0$ and $\sum_{e' \in \mathcal{E}} \delta_{e,e'} = 1$. We assume the matrix $\delta = (\delta_{e,e'} : e, e' \in \mathcal{E})$ is irreducible; i.e., for each $e, e' \in \mathcal{E}$, there exists $k \geqslant 1$ and a sequence $e_0 = e, e_1, e_2, \ldots, e_k = e'$ with each $e_i \in \mathcal{E}$ such that $\prod_{i=0}^{k-1} \delta_{e_i, e_{i+1}} > 0$. In other words, it is possible to eventually move from environment e to environment e'.

The components in the system can randomly fail and then be repaired. When the environment is $e \in \mathcal{E}$, the failure rate and repair rate of each component of type i are $\lambda_{i,e} > 0$ and $\mu_{i,e} > 0$, respectively. If there is only one environment e, i.e., $|\mathcal{E}| = 1$, then the lifetimes and repair times of components of type i are exponentially distributed with rates $\lambda_{i,e}$ and $\mu_{i,e}$, respectively. Exponential distributions are frequently used to model lifetimes of hardware and software components; e.g., see [4]. We assume that all operating components of a type i have the same failure rate $\lambda_{i,e}$ in environment e. Thus, in a system with redundancies for which not all components of a type are needed for operation of the system, the extras are "hot spares" since they fail at the same rate as the main components.

Our model includes probabilistic instantaneous cascading failures, which occur as follows. The ordered set Γ_i specifies the types of components that a failure of a type-i component can cause to simultaneously fail. When a component of type i fails, it causes a single component of type $j \in \Gamma_i$ to fail at the same time with probability $\phi_{i,j}$ (if there are components of type j up), and we call $\phi_{i,j}$ a *component-affected probability.* The events that the individual components of types $j \in \Gamma_i$ fail immediately are independent. Thus, when a component of type i fails, there are independent "coin flips" to determine which components in Γ_i fail, where the coin flip for $j \in \Gamma_i$ comes up heads (one component of type j fails) with probability $\phi_{i,j}$ and tails (no components of type j fail) with probability $1 - \phi_{i,j}$.

We allow for a cascading failure to continue as long as there are still components operational in the system. For example, the failure of a component of type i may cause a component of type j to fail (with probability $\phi_{i,j}$), which in turn makes a component of type k fail (with probability $\phi_{j,k}$), and so on. As noted in [2], the SAVE package [1] allows for only one level of cascading, but the unlimited cascading in our model makes it significantly more difficult to analyze.

We can think of a cascading failure as a tree of simultaneously failing components. The root is the component, say of type i, whose failure *triggers* the cascade. The root's children, which are from Γ_i, are those components whose immediate failures were directly caused by the root's failure. At any non-root level of the tree, these components' failures were directly caused by the failure of their parents at the previous level. Although all the failing components in a cascade fail at the same time, we need to specify an order in which they fail for our problem to be well-defined, as we explain later in Section 2.1. We assume the components in a tree fail in breadth-first order.

There is a single repairman who fixes failed components using a processor-sharing discipline. Specifically, if the current environment is e and there is only one failed component, which is of type i, then the repairman fixes that component at rate $\mu_{i,e}$. If there are b components currently failed, then the repairman allocates $1/b$ of his effort to each failed component, so a failed component of type i is repaired at rate $\mu_{i,e}/b$.

2 Markov Chain

We want to analyze the behavior of the system as it evolves over time. Because of the processor-sharing repair discipline and the exponential rates for the event lifetimes, it will suffice to define the state of the system as a vector containing the number of failed components of each type and the current environment. Thus, let $S = \{x = (x_1, x_2, \ldots, x_N, x_{N+1}) : 0 \leqslant x_i \leqslant r_i \ \forall i \in \Omega,\ x_{N+1} \in \mathcal{E}\}$ be the state space, and let $Z = [Z(t) : t \geqslant 0]$ be the continuous-time Markov time (CTMC) living on S keeping track of the current state of the system. (If we had instead considered a first-come-first-served repair discipline, then the state space would need to be augmented to keep track of the order in which current set of down components failed.) We assume that Z starts in environment $0 \in \mathcal{E}$ with no components failed, i.e., state $(0, 0, \ldots, 0)$. As noted in [2] the CTMC is irreducible and positive recurrent.

We now describe the CTMC's infinitesimal generator matrix $Q = (Q(x, y) : x, y \in S)$, where $Q(x, y)$ is the rate that the CTMC Z moves from state $x = (x_1, \ldots, x_N, x_{N+1})$ to state $y = (y_1, \ldots, y_N, y_{N+1})$. If $y_i = x_i$ for each $i \in \Omega$ and $y_{N+1} \neq x_{N+1}$, then (x, y) is an *environment transition* with $Q(x, y) = \nu_{x_{N+1}} \delta_{x_{N+1}, y_{N+1}}$. If $y_i = x_i - 1$ for one $i \in \Omega$, $y_j = x_j$ for each $j \in \Omega - \{i\}$, and $y_{N+1} = x_{N+1}$, then (x, y) is a *repair transition* corresponding to the repair of a component of type i, and $Q(x, y) = x_i \mu_{i, x_{N+1}} / (\sum_{j \in \Omega} x_j)$. If $y_i \geqslant x_i$ for all $i \in \Omega$ with $y_j > x_j$ for some $j \in \Omega$ and $y_{N+1} = x_{N+1}$, then (x, y) is a *failure transition* in which $y_i - x_i$ components of type i fail, $i \in \Omega$. Any other (x, y) with $x \neq y$ not falling into one of the above three categories is not possible, so $Q(x, y) = 0$. The diagonal entry $Q(x, x) = -\sum_{y \neq x} Q(x, y)$, as required for a CTMC; e.g., see Chapter 5 of [3].

We now determine the rate $Q(x, y)$ of a failure transition (x, y). First consider the case when cascading failures are not possible, i.e., $\Gamma_i = \varnothing$ for each type i. Then the only possible failure transitions (x, y) have $y_i = x_i + 1$ for one $i \in \Omega$, $y_j = x_j$ for each $j \in \Omega - \{i\}$, and $y_{N+1} = x_{N+1}$, and this corresponds to a single component of type i failing. Then $Q(x, y) = (r_i - x_i)\lambda_{i,x_{N+1}}$.

Cascading failures complicate the computation of $Q(x, y)$ for a failure transition (x, y). As mentioned before, a cascading failure is modeled as a tree T built from the multiset B of simultaneously failing components, where B has $y_\ell - x_\ell$ failing components of type ℓ, $\ell \in \Omega$. A tree T has a rate

$$R(T) = (r_i - x_i)\lambda_{i,x_{N+1}}\, \rho\, \eta, \tag{1}$$

where

- $(r_i - x_i)\lambda_{i,x_{N+1}}$ is the failure rate of the root (assumed here to be of type i),
- $\rho = \rho(T)$ is the product of the $\phi_{j,k}$ terms for a parent node of type j causing a child of type $k \in \Gamma_j$ to fail, and
- $\eta = \eta(T, x)$ is the product of the $1 - \phi_{j,k}$ terms from a node of type j *not* causing a component of type $k \in \Gamma_j$ to fail when there are components of type k up.

A difficulty arises since there can be many such trees corresponding to the multiset B of components failing in (x, y), and calculating $Q(x, y)$ requires summing $R(T)$ over all possible trees T that can be constructed from B. The number of such trees grows exponentially in the number of failing components in the cascade; see [2].

2.1 Example of Computing a Tree's Rate

We now provide an example of computing the rate $R(T)$ of a tree T. Let $\Omega = \{A, B, C\}$, with $r_A = r_B = r_C = 4$. Also, let $\Gamma_A = \{B, C\}$, $\Gamma_B = \{A, C\}$, and $\Gamma_C = \{A, B\}$. Suppose that $\mathcal{E} = \{0\}$, and consider the failure transition (x, y) with $x = (2, 2, 3, 0)$ and $y = (4, 4, 4, 0)$. Thus, (x, y) corresponds to 2 components each of types A and B failing and a single component of type C failing. One possible tree T corresponding to (x, y) is shown in Figure 1. We assume the nodes in T fail in breadth-first order.

The nodes depicted as double circles form the tree of failing components. The single circles correspond to components in some Γ_i but did not fail. A component type j in some Γ_i could have *not failed* because either there are components of type j up at this point but its coin flip came up tails (with probability $1 - \phi_{i,j}$), or there were no more components of type j up at this point. Each node has a label of the form i-ID, where i denotes the type of the component for that node, and ID is the position of the node in a breadth-first ordering of all the nodes (single and double circles). We include the IDs just to simplify the discussion here. We call the tree of all nodes the *supertree* corresponding to the tree T of failing nodes. The supertree is used to compute $R(T)$ of T as follows. Let u_i be the number of components of type i available in the system. Since the root is a component of type A and there are $u_A = r_A - x_A = 2$ components of type A at the start of the transition (x, y), the rate of the trigger of the cascade is $2\lambda_{A,0}$. The root then causes a component of type B to fail at node ID 2, and this occurs with probability $\phi_{A,B}$. The node at ID 3 did not fail, and at this point there are $u_C = r_C - x_C = 1 > 0$ components of type C still up, so this non-failure occurs

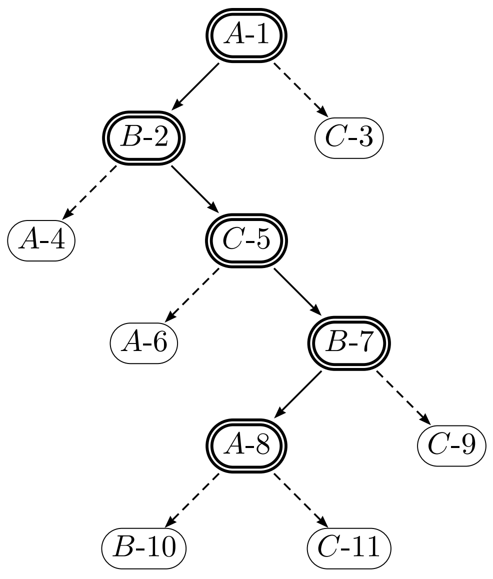

Figure 1: An example of a supertree.

with probability $1 - \phi_{A,C}$. Instead of stepping through the rest of the supertree one node at a time, we notice that all the $\phi_{i,j}$ terms must be included if T is to correspond to (x, y). Thus, in (1) we have $\rho = \phi_{A,B}\,\phi_{B,C}\,\phi_{C,B}\,\phi_{B,A}$. We notice the following when calculating η in (1):

- $1 - \phi_{i,j}$ terms are included if and only if there are still components of type j up.
- Each time we encounter a node of type j that has failed in the breadth-first enumeration of T, we decrement u_j by 1.

Keeping these observations in mind, we now calculate η. For component type A we have $u_A = 2$ before we traverse through T. As we do a breadth-first traversal through T, at ID 1, $u_A = 1$. We see that $u_A > 0$ until ID 8. So η includes terms $1 - \phi_{B,A}$ and $1 - \phi_{C,A}$ from IDs 4 and 6, respectively. For component type B we have $u_B = r_B - x_B = 2$ before we traverse through T. Since components of type B have already been exhausted at ID 7, we do not include $1 - \phi_{A,B}$ at ID 10 in η. For component type C we see that $u_C = 1$ before we traverse through T, and $u_C = 0$ at ID 5. Hence, the only contribution to η from a component of type C not failing is $1 - \phi_{A,C}$ from ID 3. Taking the product over all component types yields $\eta = (1 - \phi_{B,A})\,(1 - \phi_{C,A})\,(1 - \phi_{A,C})$. Therefore, $R(T)$ is $2\lambda_{A,0}\,\rho\,\eta$. In our implementation η is calculated through a data structure called *BFHist*, which we describe later in Section 3.1.

We previously stated that the order in which the components fail in a tree must be specified for the tree's rate to be well defined. To see why, suppose instead that the components in Figure 1 fail in depth-first order. The depth-first traversal of T is A-1, B-2, A-4, C-5, A-6, B-7, A-8, B-10, C-11, C-9, C-3. Initially, the number of components up of each type are $u_A = 2$, $u_B = 2$ and $u_C = 1$ as before. In this traversal a component of type C fails at ID 5, which makes $u_C = 0$.

Thus, η for the depth-first traversal does not include the terms $1 - \phi_{A,C}$, $1 - \phi_{B,C}$ and $1 - \phi_{A,C}$ from the subsequent type-C nodes at IDs 11, 9 and 3. In contrast, the breadth-first traversal includes one $1 - \phi_{A,C}$ term at ID 3. Hence, it is necessary to define the order in which the components fail, even though they fail simultaneously.

3 Algorithms

We now provide efficient algorithms for generating all possible trees and constructing the Q-matrix. A tree corresponds to a multiset of particular components failing, and cascading failures starting from different states can have the same multiset 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 for each corresponding transition, as was originally done in [2]. Moreover, rather than building each new tree from scratch, as was done in [2], our current algorithm builds larger trees from smaller ones already considered, leading to additional savings in the overall computational effort.

Computing the rate (1) of a given tree depends on the state from which the cascading failure began and the multiset of components that fail in the cascade. We do not actually construct the supertree in our algorithm to compute a tree's rate, but instead build a *breadth-first history* to keep track of the information necessary to compute η in (1). This is all done in Algorithms 1 (SeedTrees), 2 (AddTreeLevel) and 3 (ComputeTreeRate).

SeedTrees starts the tree generation and initializes the necessary data structures for AddTreeLevel. AddTreeLevel adds a new level to an existing tree in a recursive fashion, updates ρ in (1) to include the component-affected probabilities, and builds the tree's breadth-first history. ComputeTreeRate computes the rate of a completed tree for all the transitions it corresponds to using ρ and η computed from the breadth-first history populated in AddTreeLevel. We will now discuss each algorithm in detail. References to line numbers in the algorithms are given within angled brackets $\langle\ \rangle$.

3.1 SeedTrees

SeedTrees loops through all component types to choose a root for the tree to be constructed $\langle 1 \rangle$. One of the data structures initialized in SeedTrees is *BFHist*, which stores the breadth-first history using an array of linked lists, where the array is indexed by component type. Each linked list stores the respective parents of nodes exclusive to the supertree and stores the symbol @ for the nodes that *did* fail in the tree.

The dynamic array *level* contains the current bottom level of the tree that we are building, and it initially contains the root, which is of type *rootC.type* $\langle 5 \rangle$. We add the symbol @ to the list at *BFHist*[*rootC.type*] to denote that one component of the root's type has failed. We update *nFailed* by setting the counter for *rootC.type* to 1 $\langle 5 - 7 \rangle$. We initialize ρ in (1) to 1 since no components so far have been caused to fail by the failure of other components $\langle 8 \rangle$.

If the component type *rootC.type* 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 $\Gamma_{rootC.type} \neq \varnothing$ (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 $\langle\, 9-13\,\rangle$.

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 $rootC.type \in \Omega$ do
2:     $level$ = [ ]; {Dynamic array of failed components at tree's current bottom level}
3:     $nFailed = [0, 0, \ldots, 0]$; {Array that counts failed components of each type in the tree}
4:     $BFHist = [(\,), (\,), \ldots, (\,)]$; {Array of linked lists that keeps a history of parent component types in breadth-first order; BFHist is indexed by component types}
5:     add $rootC.type$ to $level$;
6:     $nFailed[rootC.type] = 1$;
7:     add @ to $BFHist[rootC.type]$; {Signifies one component of type $rootC.type$ has failed}
8:     $\rho = 1$; {initialize product of component-affected probabilities}
9:     if $\Gamma_{rootC.type} == \varnothing$ then
10:       ComputeTreeRate($nFailed$, $BFHist$, $\rho$, $rootC.type$);
11:    else
12:       AddTreeLevel($level$, $nFailed$, $BFHist$, $\rho$, $rootC.type$);
13:    end if
14: end for
```

3.2 AddTreeLevel

Given a tree, AddTreeLevel determines 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 the nonempty Γ_i for each of the failed components at the current bottom level. For each possible next level, we recursively call AddTreeLevel. We use $\mathcal{P}$ to denote the power set-like operation on an ordered set. In our realization of the power-set operation, components in each subset maintain their relative ordering from 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 and 0 denotes a node that is included in a Γ set but did not fail. If we have a tree where none of the leaf nodes at the current bottom level can cause any other components to fail (i.e., have empty Γ_is), we get no nextLevelPossibilities $\langle\, 1\,\rangle$.

Otherwise, we choose one possible choice for the failed components in the next level from the nextLevelPossibilities $\langle\, 2\,\rangle$. Each node in T has three attributes: a component type i; an ID, which is its breadth-first ID in the supertree; and a parentID, which is the breadth-first

ID of this node's parent in the supertree. For each node *parentC* in the current bottom level that acts as a parent, potentially causing other nodes to fail, we iterate through all of its possible children, i.e., through $\Gamma_{parentC.type}$. If any of these children actually fail, then they will be members of the set *oneNextLevelPossibility* $\langle\, 4-6\,\rangle$. 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 $\langle\, 10-12\,\rangle$.

If a child of type *childC.type* has failed (i.e., it exists within nextLevelPossibilities), the relevant data structures are updated to signify this occurrence. We increment the number of failed components of type *childC.type*, add the symbol @ to *BFHist*[*childC.type*] (to mark that a failure of this type has occurred at the current location in the tree), and update ρ by multiplying it by $\phi_{i,j}$ $\langle\, 11-13\,\rangle$. If the child does not actually fail, then we add *parentC.type* to *BFHist*[*childC.type*]. Later in ComputeTreeRate for each transition (x, y), we evaluate which of these children could have failed but did not, and which could *not* have failed because their type's redundancy was exhausted. $\langle\, 14-16\,\rangle$.

We do the above updates to the 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 recursively add another level to the current tree $\langle\, 19-23\,\rangle$. Otherwise, this tree has not changed in this pass through AddTreeLevel and we call ComputeTreeRate to compute the rate of the finalized tree $\langle\, 24-26\,\rangle$. We make sure that trees are not double counted because once a tree passes through AddTreeLevel unmodified, it is processed and discarded. We avoid making duplicate trees because each oneNextLevelPossibility is unique.

Algorithm 2 AddTreeLevel(*level*, *nFailed*, *BFHist*, ρ, *rootC.type*)

where *level* is the current level of failed components,
nFailed counts failed components by type in the tree,
BFHist is breadth-first history,
ρ is a cumulative product of component-affected probabilities,
rootC.type is the root component's type in the current tree

```
1: $nextLevelPossibilities = \bigtimes_{\substack{i=1: \\ \Gamma_{level[i]} \neq \varnothing}}^{|level|} \mathcal{P}(\Gamma_{level[i]})$;
   {Builds all possibilities for the next level (given the current level) by taking a Cartesian
   product of the power sets of non-empty $\Gamma$ sets of failed nodes in the current level}
2: for $oneNextLevelPossibility \in nextLevelPossibilities$ do
3:    $addedAChildFlag$ = False;
4:    for $parentC \in level$ do
5:       for $i \in \Gamma_{parentC.type}$ do
6:          if $\exists$ $childC \in oneNextLevelPossibility$ : $childC.type == i$ && $childC.parentID ==$
            $parentC.ID$ then
7:             $addedAChildFlag$ = True;
8:             if $nFailed[childC.type] == r_{childC.type}$ then
9:                goto line 2; {Invalid Tree, it requires more components than available}
10:            end if
```

```
11:         $nFailed[childC.type] = nFailed[childC.type] + 1$;
12:         add @ to $BFHist[childC.type]$; {One component of type $childC.type$ has failed}
13:         $\rho = \rho * \phi_{parentC.type,\, childC.type}$;
            {Update rate with appropriate component-affected probability}
14:       else
15:         add $parentC.type$ to $BFHist[childC.type]$; {One component of type $childC.type$
            has not failed, but was present in $\Gamma_{parentC.type}$}
16:       end if
17:     end for
18:   end for
19:   if $addedAChildFlag$ then
20:     AddTreeLevel($oneNextLevelPossibility$, $nFailed$, $BFHist$, $\rho$, $rootC.type$);
        {Tree can be grown further}
21:   else
22:     ComputeTreeRate($nFailed$, $BFHist$, $\rho$, $rootC.type$);
        {Current tree is complete because it cannot be grown further}
23:   end if
24: end for
```

3.3 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 since the failure rate of the root and η are transition-dependent.

Let $x' = (x_1, x_2, \ldots, x_N)$ and $S' = \{(x_1, x_2, \ldots, x_N) : 0 \leqslant x_i \leqslant r_i, \forall i \in \Omega\}$. In line 1 of ComputeTreeRate, we only loop through $x' \in S'$ and not $x \in S$ because the structure of a tree does not depend on the environment; only the tree's rate does. Also, as we explained in the example tree-rate calculation in Section 2.1, computing η in (1) requires keeping track of the number u_i of components of each type i still up as we perform a breadth-first traversal through the supertree.

Each time we encounter the symbol @ in the *BFHist*, we reduce the number available by one $\langle\, 5 - 7 \,\rangle$. If there are components still available, we update η $\langle\, 8 - 9 \,\rangle$. As soon as u_i reaches zero, we break because that type has then been exhausted and can no longer fail $\langle\, 10 - 12 \,\rangle$.

Once we have finished calculating η, we loop through all environments since the same tree can be used for transitions occurring in different environments $\langle\, 15 \,\rangle$. We convert $x' \in S'$ into a state $x \in S$ by appending an environment e $\langle\, 16 \,\rangle$. We generate a potential state y by adding *nFailed* to x' and appending environment e $\langle\, 17 \,\rangle$. Invalid states y will be generated in some instances because simply adding *nFailed* will cause some component types' number failed to exceed their redundancies $\langle\, 18 - 20 \,\rangle$. We finally compute the rate of the tree for the transition (x, y) using (1), and add this to the current cumulative rate $Q(x, y)$ for the transition $\langle 21 \rangle$.

Algorithm 3 ComputeTreeRate(*nFailed*, *BFHist*, ρ, *rootC.type*)

where *level* is the current bottom level of failed components,
nFailed counts the failed components of each type in the tree,
BFHist is breadth-first history,
ρ is a cumulative product of component-affected probabilities,
rootC.type is the root's type in the tree

```
 1: for $x' \in S'$ do
 2:    $\eta = 1$; {Cumulative product of complement probabilities of components that could
       have failed but did not}
 3:    for $i \in \Omega$ do
 4:       $u_i = r_i - x'_i$;
 5:       for parentC.type in BFHist[i] do
 6:          if parentC.type == @ then
 7:             $u_i = u_i - 1$;
 8:          else if $u_i > 0$ then
 9:             $\eta = \eta * (1 - \phi_{parentC.type,\, i})$;
10:          else
11:             break; {need $u_i > 0$ or else there cannot be any more failed nodes of type $i$}
12:          end if
13:       end for
14:    end for
15:    for $e \in \mathcal{E}$ do
16:       $x = (x', e)$;
17:       $y = (x' + nFailed, e)$;
18:       if $y$ is not a valid state then
19:          continue;
20:       end if
21:       $Q(x, y) = (r_{rootC.type} - x[rootC.type]) * \rho * \eta$;
22:    end for
23: end for
```

3.4 Computing a Tree's Rate Revisited

We now use our example in Section 2.1 to show how to compute η from (1) using the data structure *BFHist* for Figure 1. We assume the tree in Figure 1 was first constructed from several recursive calls to AddTreeLevel, which also built *BFHist* as follows.

A	@-1	*B*-2	*C*-5	@-8
B	@-2	@-7	*A*-8	
C	*A*-1	@-5	*B*-7	*A*-8

We then proceed to ComputeTreeRate to compute η. As in Section 2.1, prior to iterating through *BFHist*, we have $u_A = 2$, $u_B = 2$ and $u_C = 1$. Iteration through *BFHist* occurs as specified in ComputeTreeRate $\langle\, 3 - 14\, \rangle$.

- Iteration through the linked list at index A is as follows: The @-1 means a component of type A has failed at ID 1, so we then decrement u_A to 1. Then for the next two entries, since u_A is still positive, η includes $(1 - \phi_{B,A})$ and $(1 - \phi_{C,A})$ from B-2 and C-5 respectively in its product. Finally @-8 means a component of type A failed at ID 8, so we decrease u_A to 0.

- Iteration through the linked list at index B is as follows: @-2 and @-7 mean components of type B have failed at IDs 2 and 7 respectively, so we decrement u_B from 2 to 1 and then from 1 to 0. Since u_B is now 0, we cannot include any terms from this row in η.

- Iteration through the linked list at index C is as follows: Since u_C starts out positive, we include $(1 - \phi_{A,C})$ from A-1. @-5 means a component of type C has failed at ID 5, so we decrement u_C to 0. Since $u_C = 0$, again we cannot include any terms from this row in η.

Multiplying the contributions from each index results in $\eta = (1 - \phi_{B,A})(1 - \phi_{C,A})(1 - \phi_{A,C})$.

4 Performance Analysis

When analyzing the performance of our algorithm, it is important to note that the systems with larger numbers of states do not necessarily correspond to larger numbers of trees. When computing statistics such as Mean time to failure (MTTF) and Steady state unavailability (SSU), we rely on third-party libraries to perform efficient matrix operations. We can easily swap out libraries for these tasks so there is no need to discuss the performance of the Q-Matrix operations. When we look at the trends show in Figure 2, we see an expontentially increasing Time to Generate Trees (TGT) as the number of trees generated increases.

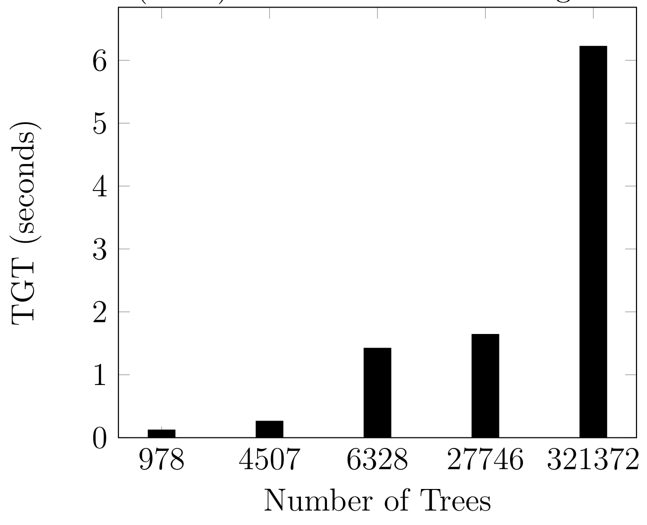

Figure 2: Relation of TGT vs Trees across various models

In an effort to improve the running time of our algorithm, we introduce certain thresholds to stop generating trees based on simple test conditions. To properly stop the recursion of the trees, we need to insert the threshold logic in Algorithm 2 before line $\langle 19 \rangle$. We will denote the new threshold pseudo-code with lines $\langle 19.x \rangle$. The breaking conditions will cause the program to continue to the next enumeration of the bottom-most tree level, thereby ignoring any rate computations of trees that do not satisfy the threshold criteria.

Our first threshold is the *height threshold* where we stop generating trees after the height of a tree reaches τ_h.

```
**if** $depth > \tau_h$ **then**
  goto line 2;
**end if**
```

The second threshold we consider is the *node threshold*, where trees will contain a maximum of τ_n components. We compute the sum of components in *nFailed* to count the number of failed components in T.

```
**if** $\sum nFailed > \tau_n$ **then**
  goto line 2;
**end if**
```

The final threshold we analyze is the *rate threshold*, where we stop generating trees after the ρ drops below τ_r.

```
**if** $\rho < \tau_r$ **then**
  goto line 2;
**end if**
```

The range of values for any τ is from 0 to ∞. However, it is important to note that τ_h and τ_n depend on variables that increase as more trees are generated. ρ decreases as more components are attached, thus τ_r needs to be zero to generate all trees, while τ_h and τ_n needs to be ∞. In order to define a more appropriate range for our thresholds, we need to define the thresholds based on each model. For initial loose bounds, we can assume the following limits: $\tau_n < \sum^{i} r_i$, $\tau_h < |LP|$ and $\tau_r < MAX(r_i\lambda_i)$.

To define the longest path (LP), we need to first describe our models as a graph data structure. The graph's nodes are the component types, where each node's value is the λ of that component type. We define the directed edges to start at the parent component type and end at the cascading component type. The value of each edge is $\phi_{i,j}$, where the edge starts at the node labeled i and ends at the node labeled j. The longest path problem is NP-hard for cyclic graphs, thus we cannot derive any finite upper bound for τ_h. It is possible to find the tallest tree in a model, however, we cannot calculate this height prior to running our algorithm.

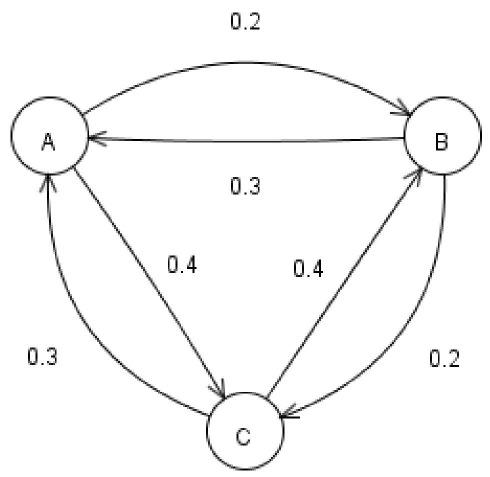

Figure 3: Graph of the 125-state model

However, we can use our graph data structure to pursue a tighter bound for the τ_r.

4.1 Iterating over thresholds

Using the model shown in Figure 3, we step through each of the thresholds. Since the results vary from model to model, we use the loose upper bounds as starting points. From there, we simply subtract τ_n by 1. τ_r starts at the largest λr and to calculate the next step we multiply by the smallest ϕ.

τ_n

N	Trees	TGT (s)	MTTF	SSU
12	321372	6.69	44.12	0.0326
11	167022	4.83	47.66	0.0291
10	63072	3.12	54.15	0.0246
9	20322	1.77	65.42	0.0198
8	6165	1.18	83.35	0.0152
7	1875	0.65	110.62	0.0152
6	588	0.41	156.08	0.0152
5	192	0.21	252.28	0.0152

τ_r

R	Trees	TGT (s)	MTTF	SSU
4E-2	31	0.09	1018.21	0.0006
8E-3	114	0.18	321.09	0.0028
16E-4	486	0.35	157.52	0.0071
32E-5	2378	0.72	98.95	0.0125
64E-6	11688	1.42	69.48	0.0186
128E-7	50396	2.83	53.93	0.0186
256E-8	163722	4.37	46.42	0.0186
512E-9	300316	6.41	44.18	0.0186

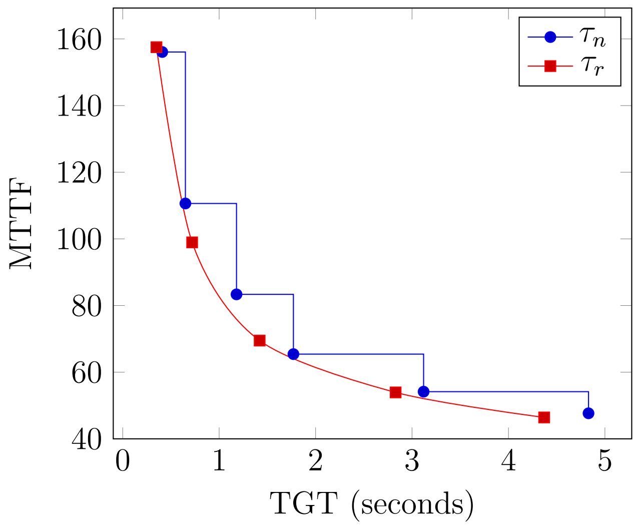

Figure 4: Comparing rate and node thresholds

For the sake of brevity, we removed certain outlying points from the graph. We used the constant left markup for the τ_n plot because the x-value represent the minimum required amount of time to achieve MTTF. The τ_r is a generic smooth line, since the precision of the rate threshold can be very minute, it will still produce some increase in MTTF. With enough points for τ_r we see a stepping effect at a smaller scale, similar to the τ_n plot.

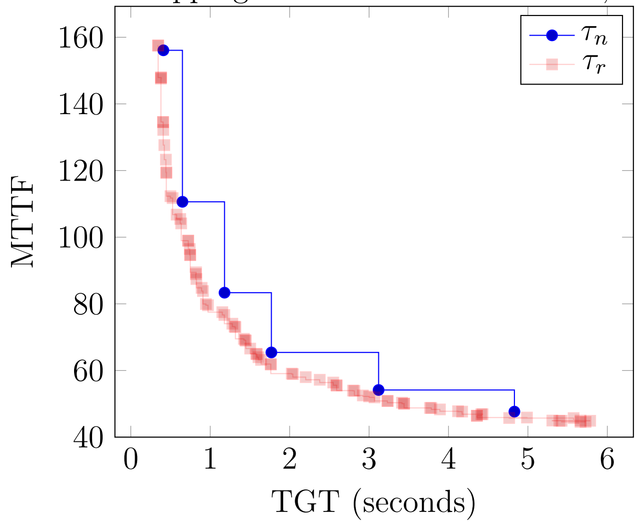

Figure 5: Smaller stepping for τ_r

Points closer to the origin of the graph represent more efficient thresholds. As shown in Figure 5, it is clear that stepping over τ_r results in better efficiency than τ_n. Furthermore, since we cannot have decimal values for τ_n, there is little reason to pursue this threshold.

4.2 Choosing an appropriate τ_r

Use Djikstra's algorithm to find the shortest path with the largest rate.

References

[1] A. Blum, P. Heidelberger, S. S. Lavenberg, M. K. Nakayama, and P. Shahabuddin. Modeling and analysis of system availability using SAVE. In Proceedings of the 23rd International Symposium on Fault Tolerant Computing, pages 137–141, 1994.

[2] S. M. Iyer, M. K. Nakayama, and A. V. Gerbessiotis. A Markovian dependability model with cascading failures. IEEE Transactions on Computers, 139:1238–1249, 2009.

[3] S. Ross. Stochastic Processes. Wiley, New York, second edition, 1995.

[4] M. Xie, Y. S. Dai, and K.L. Poh. Computing Systems Reliability: Models and Analysis. Kluwer Academic, New York, 2004.