Reliability Analysis of Tree Networks Applied to Balanced Content Replication

Mugurel Ionut Andreica, Nicolae Tapus

▶ **To cite this version:**

Mugurel Ionut Andreica, Nicolae Tapus. Reliability Analysis of Tree Networks Applied to Balanced Content Replication. IEEE International Conference on Automation, Quality and Testing, Robotics (THETA 16) (AQTR), May 2008, Cluj-Napoca, Romania. pp.79-84, 10.1109/AQTR.2008.4588711 . hal-00793908

HAL Id: hal-00793908
https://hal.science/hal-00793908

Submitted on 24 Aug 2013

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Reliability Analysis of Tree Networks Applied to Balanced Content Replication

Mugurel Ionut Andreica[1], Nicolae Tapus[1]
[1]Polytechnic University of Bucharest, {mugurel.andreica, nicolae.tapus}@cs.pub.ro

***Abstract*-The fault tolerance of the communication topology of a distributed system is a very important feature which needs to be analyzed carefully. In this paper we propose a new reliability me-tric for tree topologies, based on the unrestricted vertex multicut problem on trees, for which we present the first optimal linear time algorithm. We present evaluation results of the reliability metric on tree networks used for balanced content replication. For this problem, we also developed a new O(n·k) algorithm solving the k-equitable coloring problem on trees, based on a hierarchy of color relabeling permutations.**

I. Introduction

Fault tolerance is one of the main issues which need to be considered when designing, using and deploying distributed systems, applications and services. The communication topo-logy in particular needs to be analyzed carefully. Whether this topology is the underlying network infrastructure or the overlay graph of a peer-to-peer application, its potential fragility may have a serious impact upon the performance of the entire system. In this paper we are interested in analyzing the reliability of communication topologies having a tree structure. Trees are very fragile - the failure of a single node disconnects the network. However, because of their simplicity, trees are very useful in many domains, like multicast content distribution, packet routing, content replication and distributed data indexing and storage. In order to perform the reliability analysis, we propose a new reliability metric for trees, based on the unrestricted vertex multicut (UVMC) problem. We also present the first optimal linear time algorithm for solving this problem.

In the UVMC problem, a graph with V vertices is given, as well as H critical pairs of vertices. The problem asks for determining the minimum number of vertices which need to be removed from the graph, such that their removal disconnects every critical pair of vertices. For trees, a simple polynomial time algorithm based on computing the lowest common ancestors (LCA) of the critical pairs was given in [9]. However, the algorithm has time complexity O(V·H), which makes it un-usable for graphs with many vertices, like those arising in large scale distributed systems. In this paper, we improve that algorithm to the optimal time complexity O(V+H).

We define the reliability of a tree as the number of vertices whose removal disconnects a carefully chosen set of H critical pairs, divided by the total number of vertices belonging to at least one pair. The way critical pairs are defined depends on the purpose of the tree network. They could be pairs of vertices between which the highest amounts of traffic are recorded or pairs of vertices which, if disconnected, would highly compromise the performance of the network. We choose the situation in which tree networks are used for balanced content replication and define the critical pairs according to the specific communication patterns of this situation. First, we solved the balanced content replication problem using a new O(V·k) algorithm we developed. The algorithm solves the equitable k-coloring problem on trees with V vertices, based on a hierarchy of color relabeling permutations. We present evaluation results of the reliability metric in this case.

The rest of this paper is structured as follows. In Section II we formally define the unrestricted vertex multicut problem and present the main steps of the well known O(V·H) algorithm solving this problem on tree graphs. In Section III we present our optimal linear O(V+H) algorithm, thus improving upon the previously known best result. In Section IV we define the balanced content replication problem on trees and in Section V we present our new algorithm for equitable k-coloring of trees. In Section VI we present evaluation results, in Section VII we present related work and in Section VIII we conclude.

II. The Unrestricted Vertex Multicut Problem

We are given a connected graph G with V vertices and E edges, as well as H critical pairs (s_1,t_1), ..., (s_H,t_H). The UVMC problem asks for the minimum number of vertices which need to be removed in order to disconnect every critical pair of vertices, i.e. at least one vertex must be removed from every path between the two vertices of a critical pair; vertices belonging to critical pairs may be removed, too. When G is a tree, a simple polynomial time algorithm is the following:

Step 1. ***root*** *the tree at some vertex r and compute the parent-son relationships for all the vertices.*
Step 2. ***for*** *each critical pair* (s_i,t_i) ***do****: compute its LCA and the level of the LCA (the distance from the LCA to the root)*
Step 3. ***sort*** *all the critical pairs in non-increasing order of the level of their LCA*
Step 4. ***for*** *each critical pair* (s_i, t_i)*, in the sorted order,* ***do***
Step 4.1. ***if*** s_i *and* t_i *are not already disconnected* ***then***
Step 4.1.1. ***remove*** *the LCA of* s_i *and* t_i *from the tree*

© 20xx IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

The number of vertices removed at Step 4.1.1 is the minimum number of vertices which need to be removed in order to disconnect all the H critical pairs. The algorithm presented above can easily be implemented in time O(V·H). Step 1 only takes O(V) time. Computing the LCA of each pair in Step 2 can be done in O(V) time, so Step 2 takes O(V·H) time overall. Step 3 can be performed in O(H·log(H)) time. The connectivity test at Step 4.1 can be performed in O(V) time. Multiplying this by H, we obtain an O(V·H) complexity for Step 4.

III. A Linear Time Algorithm For The UVMC Problem On Trees

The algorithm presented in the previous section has an obvious O(V·H) implementation. However, using more intelligent techniques, it can be implemented in O(V+H) time.

Step 2 of the algorithm can be performed in time O(V+H) for all the critical pairs. In order to achieve this, we use the algorithm presented in [9]. The rooted tree is preprocessed in O(V) time. An array E containing the Euler tour of the tree traversal is produced, as well as an array L with the levels of the vertices, in the order in which they are encountered in the Euler tour. From the first array, a representative array R is computed: for each vertex u, R[u] represents the position of the first occurence of u in E. Now, in order to compute the LCA of two vertices u and v, we need to find the vertex having the minimum level and which is located between R[u] and R[v] in E. This is performed in O(1) time, by using a technique called Range Minimum Query (RMQ). The array L is first preprocessed in O(V) time, by splitting it into blocks of suitable sizes. Then, using this preprocessing, the minimum value between two given positions can be found in O(1) time. Therefore, the time complexity is O(H) for all the critical pairs. At the end of this step, we have two new arrays, *pLCA* and *level*, where pLCA[i] is the lowest common ancestor of the i^{th} pair and level[i] is the level of the i^{th} pair's LCA. All the steps presented in this paragraph are described in detail in [9] and, as mentioned, lead to an O(V+H) time complexity. Implementing the other steps in O(V+H) is the original contribution of this paper.

Step 3 of the algorithm can be implemented in O(V+H) time, by using an array of linked-lists *LL*. For each pair i, we will add the pair's index at the beginning (or at the end) of the linked-list LL[level[i]], in O(1) time. Since level[i]'s value is between 0 and V-1, the array LL only has V entries. Now, in order to sort the pairs, we will traverse the entries of LL from V-1 down to 0. If LL[i] is not empty, then we will traverse this linked-list and each element of the list is added at the end of an array *sorted_pairs*. The array sorted_pairs will contain the pairs in non-increasing order of their LCA's level. It is obvious that, because of the order of the traversals, pairs with the LCA on larger levels (located further away from the root) will be placed before pairs with the LCA on smaller levels (located closer to the root) in the sorted_pairs array. The overall complexity of this step is O(V+H). Creating the LL array takes O(V) time, inserting all the critical pairs in LL takes O(H) time and traversing the linked-lists in LL takes O(V+H) time.

Implementing Step 4 in O(V+H) time is the trickiest part of the algorithm. We will maintain an array of boolean values, *marked*. For each vertex of the tree, this array will tell if the vertex was marked or not. Initially, no vertex is marked. We will consider the critical pairs in the order produced at Step 3. Checking if the two vertices of the pair are disconnected will be done in O(1), by simply inspecting the marked array. If at least one of the two vertices was marked, then the two vertices were disconnected because of the removal of the LCA of a previous pair. If they are still connected, we will "remove" and mark their LCA, as well as mark all the unmarked vertices located in their LCA's subtree. This time, removing a vertex from the tree does not mean deleting it from the tree, together with the incident edges. The tree is not modified, only a counter with the number of "removed" vertices is increased.

Let's first analyze the correctness of this algorithm. The part that needs to be considered is the connectivity test for the two vertices s_i and t_i of a critical pair i. For this, we use Theorem 1.

Theorem 1: *When considering a critical pair (s_i,t_i) in Step 4 of the algorithm and (marked[s_i]=True or marked[t_i]=True), then the two vertices of the pair have already been disconnected.*

Proof. Without loss of generality, we will consider that vertex s_i is marked (t_i could be marked, too). If s_i is marked, this is because some ancestor x of s_i (a vertex on the path from s_i to the root of the tree) was removed and this lead to all the vertices in x's subtree being marked. This ancestor x was the LCA of the vertices of a pair considered in Step 4 before pair i. Because the levels of the LCAs of the pairs are sorted in non-increasing order, vertex x's level must be greater than or equal to level[i]. Note that all the ancestors of s_i whose level is greater than or equal to level[i] are located on the path between s_i and pLCA[i] (including the endpoints of the path, too). Therefore, we conclude that at least one vertex on the path between s_i and pLCA[i] was removed previously. Now it is easy to prove that the vertices s_i and t_i are disconnected. In the tree, the path from s_i to t_i is unique. We will consider this path as being composed of two parts: the path from s_i to pLCA[i] and the path from pLCA[i] to t_i. Since we know that at least one vertex on the path from s_i to pLCA[i] was removed previously, this means that at least one vertex on the path from s_i to t_i was removed, which concludes our proof.

Theorem 1 proves the correctness of the algorithm, but the time complexity of Step 4 is not obvious, yet. There could be O(V) vertices removed and each of them may have O(V) unmarked vertices in its subtree, which would make the time complexity $O(V^2+H)$. This is where we use Theorem 2:

Theorem 2. *If some vertex v of the tree is marked, then all the vertices in v's subtree are marked, too.*

Proof. If v is the LCA of some pair which is removed in order to disconnect the vertices of that pair, then we will mark v and all the unmarked vertices in its subtree. Therefore, all the vertices in v's subtree will be marked. If v is not one of the removed vertices, then v was marked because of the removal of some ancestor x of v. When vertex x was removed, all the vertices in x's subtree were marked. Since all the vertices in v's subtree also belong to x's subtree, they were marked, too, and this concludes the proof.

Using Theorem 2, we can use the recursive algorithm **TraverseAndMark**, presented below, for marking the unmarked vertices in the subtree of a vertex v. In this algorithm we denote by $sons_v$ the set of sons of the vertex v.

TraverseAndMark(v):

```
marked[v]=True
for each w ∈ sons_v do
  if (not marked[w]) then
    TraverseAndMark(w)
```

TraverseAndMark marks only the unmarked vertices in the subtree of the vertex v given as an argument. Since no vertex is marked twice, TraverseAndMark is called at most V times during Step 4 of the algorithm. During each call, all the sons of the vertex v given as an argument are considered. Overall, all the calls do not take more time than calling TraverseAndMark once for the root of the tree, which takes O(V) time. Therefore, Step 4 of the algorithm has time complexity O(V+H). The pseudocode of the whole algorithm is given below:

UVMC(tree with V vertices, H pairs (s_1,t_1), ..., (s_H,t_H)):

Step 1: Choose a root r and compute the parent-son relationships for all the vertices of the tree.

Step 2: Compute the arrays pLCA and level: pLCA[i] is the lowest common ancestor of the i^{th} pair and level[i] is the level of their LCA in the tree.

Step 3:

```
for l=0 to V-1 do
  LL[l]=empty
for i=1 to H do
  LL[level[i]].add(i)
sorted_pairs=empty
for lev=V-1 downto 0 do
  if (LL[lev] is not empty) then
    for i in LL[lev] do
      sorted_pairs.add(i)
```

Step 4:

```
for v=1 to V do
  marked[v]=False
num_removed=0
for i=1 to H do
  p=sorted_pair[i]
  if ((not marked[s_p]) and (not marked[t_p])) then
    num_removed = num_removed + 1
    TraverseAndMark(pLCA[p])
return num_removed
```

IV. The Balanced Content Replication Problem On Trees

Replication of content is a common technique employed both for reliability and load balancing purposes. In the ba-lanced content replication problem, we are given k pieces of content of equal importance, which have to be placed in the V vertices of a tree network. Within each vertex, only a single piece of content can be placed. For each piece of content i ($1 \le i \le k$), we define nv_i as the number of vertices in which the piece was placed. Because each piece is of equal importance, the number of vertices into which two different pieces are placed should be approximately equal. More formally, $|nv_i - nv_j| \le 1$, for any two pieces of content i and j.

Each vertex of the tree is a server used both for storing a re-plica of some piece of content and for serving client requests. A client may require any piece of content and the server will get that piece from the nearest server possessing it. In order to minimize network traffic, it is desirable to find a replica of the required piece of content very close to the server. In particular, the traffic is kept low if either the server possesses that piece of content or one of its neighbors does. In order to maximize the chance that one of its neighbors possesses a piece of content that the server does not possess, any two neighboring servers should not store the same piece of content.

The problem translates into an **equitable coloring** of the tree network, using exactly k colors. A supplementary assumption that we consider is that the number of pieces is not smaller than the maximum number of neighbors a server has, i.e. that the number of colors is greater than or equal to the maximum degree of any vertex of the tree.

V. A Greedy Algorithm For The K-Equitable Coloring Problem On Trees

We will start with some definitions. If an edge (i,j) belongs to the tree, then i is a neighbor of j and j is a neighbor of i. The degree deg_i of a vertex i is equal to the number of its neighbors. The maximum degree of the tree is:

$$D = \max_{\text{i is a vertex in the tree}} \{deg_i\} \qquad (1)$$

If the tree has one or two vertices, then finding an equitable coloring is trivial. Another trivial situation is if the number of colors k is greater than or equal to V, because in this case, each vertex can be colored with a different color. Therefore, we will only consider the case $V \geq 3$ and $k < V$ (and $k \geq D$).

We will transform the tree into a rooted tree, by choosing a vertex r as the root. This vertex can be any vertex whose degree is less than D. For $V \geq 3$ vertices, such a vertex can always be found (for instance, the root can be any vertex of degree 1, because the maximum degree D is greater than 1). Considering the rooted tree, each vertex has a parent (except the root) and all of its neighbors except for its parent become its sons. Vertex i has ns(i) sons. Each vertex of degree D has D-1 sons, which is the maximum number of sons any vertex may have. In an equitable coloring of a subtree or of a forest, we will call c a *surplus* color if there is one extra vertex colored with c, compared to the color having the minimum number of vertices colored with it. If the total number of vertices in the subtree (forest) is divisible by k, there will be no surplus colors.

In the first stage, the algorithm will compute several values for each vertex i of the tree, in a bottom-up fashion (from the leaves towards the root). Calling *GreedyEquitableColoringPhase1(r, k)* achieves this. The following values will be computed (we denote by *A mod B* the remainder of the integer division of A and B):

- $nvtotal_i$ = the number of vertices in i's subtree (including i).
- $ncplus_i = (nvtotal_i \bmod k)$ – the number of surplus colors in an equitable coloring of vertex i's subtree.
- $color_i$ = the color of vertex i in an equitable coloring of its subtree.
- $color\_perm_i$ = a permutation which describes how the colors in vertex i's subtree should be relabeled.

During the first traversal of the tree, the actual colors of the vertices are not fully computed. For each vertex i, we will know its color in an equitable coloring of the subtree rooted at i. This color will not necessarily be the final color of the vertex, because some of its ancestors might choose to relabel the colors in i's subtree. Relabeling colors is the main mechanism employed by our algorithm. The relabeling is described as a permutation *p*, where the values $y=p(x)$ have the meaning that if a vertex was assigned color x, then it will be reassigned color y. The *color_perm* permutations form a hierarchy of relabeling permutations. The actual color of vertex i will be obtained by first composing all the *color_perm* permutations on the path from the root to vertex i into a permutation p and then assigning to vertex i the color $p(color_i)$. For instance, if the path from the root r to the vertex i is composed of the vertices $r=v_1, v_2, ..., v_q=i$, then $p=color\_perm_{v1} \bullet color\_perm_{v2} \bullet \ldots \bullet color\_perm_{vq}$. We will show next how to compute all the values mentioned above, especially the *color_perm* permutations, which will be used in a second top-down traversal of the tree.

If i is a leaf, then i's color will be set to 1 and the *color_perm* permutation will be set to the identity permutation (1,2,..,k). If i is not a leaf, then an equitable coloring will be found for the subtree rooted at each son of i, independently. In order to combine all the colorings of the subtrees of i's sons, some colors will have to be relabeled, i.e. for some of the sons, the *color_perm* permutation will need to be changed. An entry $color\_perm_j(c)$ means that every vertex that was colored with color c in j's subtree will need to be recolored with the color $color\_perm_j(c)$. Obviously, relabeling the colors in a subtree will not change the equitable coloring of that subtree (the actual colors of the vertices will be changed, but the difference between the number of vertices colored with any two distinct colors will still remain at most 1).

The computation of an equitable coloring for the subtree rooted at a vertex i will maintain the following invariant: the $ncplus_i$ surplus colors will be the colors $1,2,...,ncplus_i$. Vertex i's color will be $ncplus_i$, if $ncplus_i > 0$, or 1, otherwise.

We will now explain how to combine the equitable colorings of the subtrees rooted at vertex i's sons into an equitable colo-ring of vertex i's subtree. We will consider vertex i's sons in some arbitrary order $s_{i,1}, s_{i,2}, ..., s_{i,ns(i)}$ (we will denote $s_{i,j}$ by s(i,j), too). After considering all the ns(i) sons, their colors will belong to the set $\{1,\ldots,k-1\}$, so that it will be possible to assign color k to vertex i. Furthermore, the color k will not be a surplus color, so assigning color k to vertex i will lead to an equitable coloring of vertex i's subtree. After assigning color k to vertex i, we will change $color\_perm_i$ accordingly, in order to maintain the invariant.

After considering the first j-1 sons, the number of surplus colors will be:

$$cplus_{j-1} = \left(\sum_{p=1}^{j-1} nvtotal(s_{i,p}) \right) \bmod k \qquad (2)$$

Moreover, the colors $\{1,2,..,cplus_{j-1}\}$ will be the surplus co-lors. When reaching the j^{th} son, each of the first j sons is in one of the following two states: *active* or *inactive*. If $ncplus_{s(i,j)}=0$, then $s_{i,j}$ is inactive, otherwise $s_{i,j}$ is active. If ($ncplus_{s(i,j)}>0$ and $cplus_{j-1}=0$) or ($cplus_{j-1} + ncplus_{s(i,j)}>k$), then all the sons $s_{i,1},\ldots, s_{i,j-1}$ are made inactive and $s_{i,j}$ will be the only active son. A counter c_{active} is maintained, storing the number of currently active sons. If the j^{th} son is active, the colors in its subtree will be permuted in a cyclic manner, such that color c ($1 \leq c \leq k$) becomes color $((cplus_{j-1}+c-1) \bmod k)+1$ (this change is applied to the $color\_perm_{s(i,j)}$ permutation, in order to maintain the invariant). Then, $s_{i,j}$'s subtree is added to the forest composed of the subtrees of the first j-1 sons.

After that, $s_{i,j}$'s color will be relabeled (whether it is active or not), according to some rules we will mention in the following paragraph. If $s_{i,j}$ is active, this relabeling needs to be "visible" to all the previous sons, but must not be "visible" to the sons which were not considered yet, i.e. we must also relabel the color classes of the previously considered sons, but not those of the sons which were not yet considered. This can be achieved by applying the relabeling directly to the $color\_perm_{s(i,p)}$ permutations of every son $s_{i,p}$ ($p \leq j$), but this would lead to a $O(V^2 \cdot k)$ algorithm. Instead, we will maintain a stack of rela-beling permutations. Then, after considering all the sons, we will need to compose all the permutations on the stack, from the top down to each level *lev* into a permutation p_{lev} and then replace $color\_perm_{s(i,lev)}$ by $p_{lev} \bullet color\_perm_{s(i,lev)}$, for $1 \leq lev \leq ns(i)$. All the p_{lev} permutations can be computed in $O(k \cdot ns(i))$ time overall, so this maintains the $O(V \cdot k)$ complexity of the algorithm.

As stated in the previous paragraph, $s_{i,j}$'s color will be relabeled. If $s_{i,j}$ is an inactive son, then we will swap its color with color k-1. This swap will be represented as a relabeling permutation and can be applied to the $color\_perm_{s(i,j)}$ permutation only. The swap does not need to be visible to any of the other sons. Therefore, we will apply the swap to the $color\_perm_{s(i,j)}$ permutation and push on the stack the identity permutation.

If $s_{i,j}$ is an active son, we would like to swap $s_{i,j}$'s color with the color indicated by the counter c_{active}. This can also be achieved using a simple relabeling permutation, which swaps the two color classes. However, this could cause some pro-blems, for instance, if $s_{i,j}$'s color is k, because then the color c which will be relabeled to k might have been assigned to some other son. If this happens, it will be impossible to assign color k to vertex i in the end and the algorithm will be incorrect. The solution, however, is simple. We will swap not just two colors, but three. We will swap $s_{i,j}$'s color, the color c_{active} and some color *auxcol* which was not assigned to any previous son. These swaps can be described by a relabeling permutation containing a cycle of length three (formed by the three colors). Finding a color *auxcol* not assigned to any previous son is easy: we will maintain a counter $total_{active}$, denoting the total number of sons which have ever been active (including $s_{i,j}$). Then, the color $total_{active}$ is just the color we need. The relabe-ling permutation will be pushed on the stack, as it needs to be visible to all the sons $s_{i,p}$ ($p<j$). This relabeling with three colors will be used only if vertex $s_{i,j}$'s color is greater than the value of $total_{active}$; in the other situation, we will swap $s_{i,j}$'s color and the color c_{active} directly (without using an auxiliary color).

After adding all of vertex i's sons, the obtained forest is equitably colored, the colors of vertex i's sons belong to the set {1,..,k-1} and the first $cplus_{ns(i)}$ ($0 \leq cplus_{ns(i)} < k$) colors are the surplus colors. By assigning the color k to vertex i, the coloring is kept equitable and valid. All that remains to be done is to relabel vertex i's color with $cplus_{ns(i)}+1$, in order to maintain the invariant that the $ncplus_i$ surplus colors in an equitable co-loring of i's subtree are the colors 1,2,...,$ncplus_i$. This is accomplished by swapping the colors k and $cplus_{ns(i)}+1$ in the $color\_perm_i$ permutation.

In order to find the actual color of each vertex, we will have to traverse the tree again, starting from the root (in a top-down fashion this time). We will maintain a stack of coloring permutations. The first permutation pushed on the stack will be the identity permutation. When going from some vertex i to one of its sons s(i,j), we will compose the color permutation on the top of the stack with $color\_perm_{s(i,j)}$ and push this permutation on the stack. The permutation at the top of the stack will then be used for finding the real color of vertex s(i,j). When returning from a son s(i,j) to its parent i, the permutation from the top of the stack is popped. It is easy to notice that both parts of the algorithm take $O(V \cdot k)$ time. The pseudocode is showed below:

GreedyEquitableColoringPhase1(i, k):

```
if (ns(i)=0) then
  color_perm_i=(1,2,..,k) // the identity permutation
  color_i=nvtotal_i=1
  ncplus_i=1 mod k
  return
// at first, find an equitable coloring for each of vertex i's sons
for j=1 to ns(i) do
  GreedyEquitableColoringPhase1(s_{i,j}, k)
// combine the equitable colorings of vertex i's sons
nvtotal_i=cplus_0=c_active=total_active=0
stack=empty
for j=1 to ns(i) do
  nvtotal_i=nvtotal_i+nvtotal_{s(i,j)}
  if (ncplus_{s(i,j)}=0) then
    Swap2(s_{i,j}, k-1, k)
    stack.push((1,2,..,k))
  else
    CyclicPermutation(s_{i,j}, cplus_{j-1}, k)
    c_active=c_active+1
    total_active=total_active+1
    if (cplus_{j-1}+ncplus_{s(i,j)}>k) then
      c_active=1
    soncolor=color_perm_{s(i,j)}(color_{s(i,j)}).
    if (soncolor>total_active) then
      stack.push(Swap3(s_{i,j}, c_active, total_active, k))
```

```
    else
      perm=(1,2,..,k)
      perm(soncolor)=c_active
      perm(c_active)=soncolor
      stack.push(perm)
    if (cplus_j-1+ncplus_s(i,j)=k) then
      c_active=0
  cplus_j=nvtotal_i mod k
// empty the stack
lev=ns(i)
p_lev+1=(1,2,..,k)
while (not stack.isEmpty()) do
  p_lev=ComposePermutations(p_lev+1, stack.top(), k)
  color_perm_s(i,lev)=ComposePermutations(p_lev,color_perm_s(i,lev),k)
  stack.pop()
  lev=lev-1
// choose a color for the vertex i
color_i=k
color_perm_i=(1,2,..,k)
nvtotal_i=nvtotal_i+1
ncplus_i=(cplus_ns(i)+1) mod k
Swap2(i, cplus_ns(i)+1, k) // relabel vertex i's color with cplus_ns(i)+1
```

CyclicPermutation(j, offset, k):

```
for c=1 to k do
  color_perm_j(c)=((color_perm_j(c)+offset-1) mod k) + 1
```

Swap2(j, newcol, k):

```
oldcol=color_perm_j(color_j)
find c' such that color_perm_j(c')=newcol
color_perm_j(c')=oldcol
color_perm_j(color_j)=newcol
```

Swap3(j, newcol, auxcol, k):

```
perm=(1,2,..,k)
oldcol=color_perm_j(color_j)
perm(oldcol)=newcol
perm(newcol)=auxcol
perm(auxcol)=oldcol
return perm
```

ComposePermutations(p_1, p_2, k):

```
for c=1 to k do
  p_result(c)=p_1(p_2(c))
return p_result
```

GreedyEquitableColoringPhase2(i, k, stack):

```
real_color_p = ComposePermutations(stack.top(), color_perm_i, k)
stack.push(real_color_p)
real_color_i = real_color_p(color_i)
for j=1 to ns(i) do
  GreedyEquitableColoringPhase2(s_i,j, k, stack)
stack.pop()
```

VI. Evaluation Results

We considered two types of test scenarios. For the first type, we chose different values for the following parameters: the number of vertices of the tree V, the number of colors (pieces of content) k, the maximum degree D (because k≥D) and the number of leaf vertices. Then, using the algorithm presented in Section V, we equitably colored the tree (we chose a balanced distribution of content replicas). We then chose an extra parameter C, representing the number of vertices used for serving client requests; the actual vertices were then chosen randomly. The critical pairs were the pairs of vertices (i,j) where i is a vertex serving client requests and j is one of the closest vertices to i, colored with its color (other vertices colored with j's color are farther away from i). The results are showed in Table I.

It is clear that the reliability decreases with the number of colors (for fixed V and D) and with the maximum degree (for fixed V and k), with a few minor exceptions. We also studied reliability variations for fixed V, variable leaf percentage and variable number of colors. Fig. 1 shows that reliability decreases as the leaf percentage increases. Furthermore, we tried to understand how

the reliability would change with the number of vertices and different leaf percentages. Fig. 2 shows that leaf percentage matters much more than the number of vertices.

In the second type of test scenarios, we considered critical pairs of the form (i,j), where i and j are two vertices with the same color. The motivation behind this was that different replicas need to be synchronized occasionally, so communication between vertices hosting the same replica is needed. We obtained results which are similar to the ones for the first type of test scenarios. The experiments showed that tree structure, rather than other parameters, is the most important in terms of reliability. This corresponds to our expectations, so we can conclude that the reliability metric is indeed representative.

We also implemented an $O((V+H)\cdot\log(V))$ version of our unrestricted vertex multicut algorithm and compared it to the $O(V\cdot H)$ solution (see Table II).

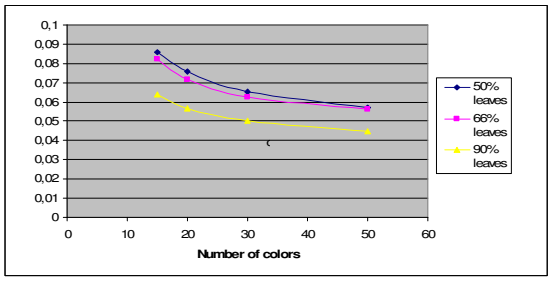

Figure 1. Variation of reliability values for V=10000 vertices.

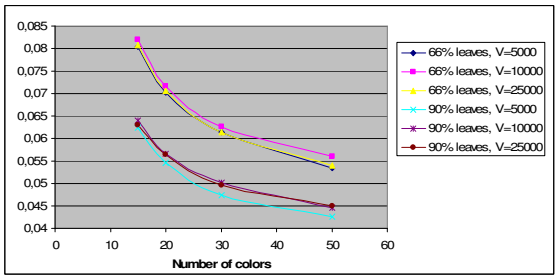

Figure 2. Reliability metric for different values of V and leaf percentages.

TABLE I

RESULTS FOR THE FIRST TYPE OF TEST SCENARIOS

V	D	k	C	H	removed vertices	vertices in pairs	reliability	number of leaves
10000	2	2	1000	2000	987	2705	0,3649	2
10000	2	3	1000	2000	987	2705	0,3649	2
10000	2	5	1000	4000	987	4101	0,2407	2
10000	2	10	1000	9792	987	6550	0,1507	2
10000	3	3	1000	2774	954	3215	0,2967	50%
10000	3	4	1000	3787	954	3835	0,2488	50%
10000	3	5	1000	4987	954	4488	0,2126	50%
10000	3	10	1000	10587	954	6599	0,1446	50%
10000	5	5	1000	5545	935	4525	0,2066	50%
10000	5	6	1000	6884	935	5054	0,1850	50%
10000	5	10	1000	11885	935	6412	0,1458	50%
10000	15	15	500	9557	485	5642	0,0860	50%
10000	15	20	500	12981	485	6410	0,0757	50%
10000	15	30	500	19446	485	7412	0,0654	50%
10000	15	50	500	31735	485	8511	0,0570	50%

TABLE II

COMPARISON OF UVMC ALGORITHMS (PYTHON IMPLEMENTATION)

V	H	O(V·H) algorithm	O((V+H)·log(V)) algorithm
10000	9998	10.25 sec	0.37 sec
30000	29999	86 sec	1.39 sec
50000	50001	229.58 sec	2.18 sec
66666	66666	1149.22 sec	3.70 sec
99999	99998	1896.78 sec	5.71 sec

We implemented an $O((V+H)\cdot\log(V))$ algorithm for determining the lowest common ancestors [10], instead of the $O(V+H)$ algorithm, which we considered too complicated for practical use. The LCA algorithm computes for each vertex i a table Anc[i][j]=the ancestor of vertex i which is 2^j levels upwards. Anc[i][0] = the parent of vertex i and Anc[i][j] = Anc[Anc[i][j-1]][j-1] ($j\geq 1$). With this table, the LCA of two vertices can be determined in $O(\log(V))$ time. We implemented the algorithms in Python and we ran them on an Intel Core 2 Duo processor, with 1 GB of RAM. The practical results confirmed the improvements predicted by the theoretical results.

VII. Related Work

Reliability metrics have been proposed before in many research papers [1,2], but, as far as we are aware, none of them uses the unrestricted vertex multicut as a subproblem for computing the metric's values. The O(V·H) solution for the Unrestricted Vertex Multicut Problem on trees was presented in [3]. Other papers [4] studied the vertex multicut problem, but their focus was on making a distinction between classes of problems which are solvable in polynomial time, and not on developing efficient polynomial time algorithms, like we did in this paper.

Equitable colorings of trees have been studied either expli-citly [5,6] or by solving scheduling problems [7]. In [5], the equitable coloring of trees with a minimum number of colors was studied and a polynomial time algorithm was proposed. In [6], the authors present an $O(n^3)$ algorithm for the equitable k-bounded vertex coloring of trees with n vertices. In [7], the authors try to minimize the total number of colors used, subject to limitations like the maximum number of vertices colored with the same color and they present a linear time algorithm for trees, but are not necessarily interested in obtaining an equitable coloring. Equitable colorings of graphs with bounded treewidth have also been studied [8], but so far the known polynomial algorithms are only of theoretical interest.

VIII. Conclusions And Future Work

This paper brings several original theoretical and practical contributions, as follows: a new reliability metric for analyzing tree networks, the first linear time algorithm for the unrestricted vertex multicut problem on trees and a new algorithm for the equitable k-coloring problem on trees. The reliability metric was evaluated on tree networks used for balanced content replication and was found to be of significant interest.

As part of our future work, we intend to evaluate the reliabi-lity metric on tree networks used for point-to-point and multicast content distribution. Evaluations of practical examples (peer-to-peer overlays and physical network infrastructures) are a priority and will be considered in the future.

References

[1] A. P. Wood, "Reliability-metric varieties and their relationships," *Proceedings of the IEEE Reliability and Maintainabilty Symposium*, pp. 110-115, 2001.

[2] G. E. Weichenberg, V. W. S. Chan, M. Medard, "High-Reliability Architectures for Networks under Stress," *Proceedings of the 24th Annual Joint Conference of the IEEE Computer and Communications Societies*, 2004.

[3] J. Guo, F. Huffner, E. Kenar, R. Niedermeier, J. Uhlmann, "Complexity and Exact Algorithms for Multicut," *Proceedings of the 32nd International Conference on Current Trends in Theory and Practice of Computer Science, LNCS*, vol. 3831, pp. 303-312, 2006.

[4] G. Gottlob, S. T. Lee, "A logical approach to multicut problems," *Information Processing Letters*, vol. 103(4), pp. 136-141, 2007.

[5] B.-L. Chen, K.-W. Lih, "Equitable coloring of trees," *J.Combin.Theory Ser B*, vol. 61, pp. 83–87, 1994.

[6] M. Jarvis, B. Zhou, "Bounded vertex coloring of trees," *Discrete Math.*, vol. 232, pp. 145–151, 2001.

[7] B. Baker, E. G. Coffman, Jr., "Mutual exclusion scheduling," *Theoretical Computer Science*, vol. 162(2), pp. 225–243, 1996.

[8] H. L. Bodlaender, F. V. Fomin, "Equitable colorings of bounded treewidth graphs," *Theoretical Computer Science*, vol. 349(1), pp. 22–30, 2004.

[9] M. A. Bender, M. Farach-Colton, "The LCA Problem revisited," *Proceedings of the Latin American Symposium on Theoretical Informatics, LNCS*, vol. 1776, pp. 88-94, 2000.

[10] M. A. Bender, M. Farach-Colton, "The Level Ancestor Problem Simplified," *Theoretical Computer Science*, vol. 321, pp. 5-12, 2004.