In the 1970s, Engel devised the Stochastic Abacus as a way to compute probabilities for certain discrete probability problems with minimal numerical computation [2]. More recently, Torrence used the same technique to determine winning probabilities for players in the game “Pass the Buck” for a variety of families of graphs [7]. The Stochastic Abacus has found more widespread exposure due to a recent article by Propp in Math Horizons [4]. Levasseur analyzed Pass the Buck for complete binary trees, making use of the symmetry of these trees at all levels [3]. One such tree is shown in Figure 1. In this note, we use similar logic to describe how the game on an arbitrary rooted tree can be analyzed, making it possible to anticipate the outcome for more complex trees. One such example is the rooted tree in Figure 2. Note that the roots of all rooted trees are drawn here with roots on the top.

**Figure 1** A complete binary tree to level three.

**Pass the Buck**

“Pass the Buck” is a game of pure chance played on a connected undirected graph, with a distinguished “start vertex.” It was introduced as the end-game to a dice game called *Left Center Right* by B. Torrance and R. Torrance with a cycle graph [6]. It was later generalized by B. Torrence [7]. The game proceeds in steps starting with the start vertex holding a prize (the “buck”). A player is presumed to be at each vertex. The vertex and the player at that vertex are referred to interchangeably here. Throughout this paper, the start vertex will be the root vertex of the tree. At every stage of the game, the vertex that currently holds the buck and its neighboring vertices are selected randomly and uniformly. If the current vertex is selected, then the game ends with that vertex winning. If a neighboring vertex is selected, then the buck is passed there and the process is repeated. More precisely, if the degree of the vertex that holds the buck is $k$, then the buck moves to any of the neighbors with probability $1/(k+1)$ and the game ends with the player at the current vertex winning with probability $1/(k+1)$.

*Math. Mag.* 96 (2023) 446–454. doi:10.1080/0025570X.2023.2231859 © Mathematical Association of America

MSC: 05C85, 60C05
The Stochastic Abacus

Engle’s Stochastic Abacus is a chip-firing algorithm that, when complete, allows computation of winning probabilities for Pass the Buck. Chip-firing algorithms are applied to directed graphs with vertices that contain variable numbers of chips (or coins or M & M’s). The chips move around by the process of chip-firing. This can happen if a vertex contains enough chips to distribute along all of its outgoing edges. What makes the algorithm work is that these firings mimic random events. In the case of Pass the Buck, if a vertex has enough chips for its neighbor and itself, then giving one to each neighbor and keeping one for itself captures all possible outcomes when the buck happens to be held by that vertex. In the course of the whole stochastic abacus algorithm, the number of firings at any vertex is proportional to the number of times we expect that vertex to have an opportunity to win the game. We will illustrate how the algorithm is set up and executed for the rooted tree in Figure 3.

The abacus for Pass the Buck is constructed by first converting the graph on which it is played from an undirected to a directed graph. Each undirected edge is replace with a pair of directed edges, one in each direction. The vertices of the original graph are referred to as the “internal vertices.” Then we add “absorbing vertices” to the graph, one for each internal vertex, and an edge leading into each absorbing vertex from its corresponding internal vertex, as in Figure 4. For this example, we use the notation \( v_x \) for an internal vertex and \( t_x \) for its corresponding absorbing vertex. Absorbing vertices
are so-called because they capture (or absorb) chips in the process of chip-firing, as discussed below.

Now that we have constructed the graph for the abacus, we distribute chips to each internal vertex. The internal vertices correspond to the vertices of the original game graph, and the number of chips each gets is one less than the outdegree of the vertex in the abacus. In Figure 4, the number of chips given to each vertex is indicated inside the vertex. Initially, the absorbing vertices get no chips. The starting location of the “buck” in Pass the Buck is the starting vertex of the algorithm. In all of our examples, the starting vertex is the root of the tree. At this point in the algorithm, we say that the system is “critically loaded.” The algorithm then consists of repeatedly adding a chip to the start vertex, \( v_{\text{root}} \), and then repeatedly “firing” chips whenever the number of chips in an internal vertex is greater than or equal to the outdegree of that vertex. Firing at a vertex involves distributing a chip along each outgoing edge of the “loaded vertex” to neighboring vertices, including its own absorbing vertex. When no more firing can take place, we examine the contents of the internal vertices. If the chip counts in all internal vertices match the critical loading condition, then we end the algorithm. Otherwise, we repeat the process, adding a single chip to the starting vertex.

![Figure 4](image.png)

**Figure 4** The Stochastic Abacus for a complete level 1 binary tree. Vertices labeled \( v_x \) are internal vertices. Vertices labeled \( t_x \) are absorbing vertices.

The remarkable fact is that after we have returned to the original critical loading of internal vertices, the probability that any vertex wins the game is equal to the number of chips in its corresponding absorbing vertex divided by the total number of chips in all absorbing vertices. To see why this is plausible, consider that the process of firing chips at a vertex simultaneously simulates all the possibilities in the game when the buck reaches that vertex. Moving a chip to the absorbing vertex captures the situation where that vertex keeps the buck, while the chips that move to adjacent internal vertices capture the situations where the buck is passed to those vertices. The number of times any vertex has an opportunity to fire is proportional to its probability of winning the buck. For a complete proof, see Snell [5].

Table 1 displays a step by step account of how the process plays out in our example, with the numbers in each vertex’s column being the number of chips it holds.

Notice that after step 8, the added chip at the root still does not allow for any firing. The algorithm dictates that we test for critical loading, which is not the case at this point. However, adding yet another chip at the root in step 9 completes the algorithm.

It may not be obvious, but if two vertices can fire, as is the case before steps 4 and 5, the order in which they are fired does not matter. See Bjöner for a proof [1]. After
TABLE 1: The steps in performing the stochastic abacus on a complete binary tree to level 1.

| Step | Comment        | $v_{\text{root}}$ | $v_L$ | $v_R$ | $t_{\text{root}}$ | $t_L$ | $t_R$ |
|------|----------------|-------------------|-------|-------|-------------------|-------|-------|
| 1    | Critically loaded | 2                 | 1     | 1     | 0                 | 0     | 0     |
| 2    | Add 1 chip to $v_{\text{root}}$ | 3                 | 1     | 1     | 0                 | 0     | 0     |
| 3    | $v_{\text{root}}$ fires | 0                 | 2     | 2     | 1                 | 0     | 0     |
| 4    | $v_L$ fires       | 1                 | 0     | 2     | 1                 | 1     | 0     |
| 5    | $v_R$ fires       | 2                 | 0     | 0     | 1                 | 1     | 1     |
| 6    | Add 1 chip to $v_{\text{root}}$ | 3                 | 0     | 0     | 1                 | 1     | 1     |
| 7    | $v_{\text{root}}$ fires | 0                 | 1     | 1     | 2                 | 1     | 1     |
| 8    | Add 1 chip to $v_{\text{root}}$ | 1                 | 1     | 1     | 2                 | 1     | 1     |
| 9    | Add 1 chip to $v_{\text{root}}$ | 2                 | 1     | 1     | 2                 | 1     | 1     |

step 9, the three internal vertices are back to being critically loaded, and the process ends. We can now determine the win probabilities for each vertex. It is the number of chips in its corresponding absorbing vertex divided by the total number of chips in all absorbing vertices. For this example, the total number of chips in the absorbing vertices is 4 and the root has 2, so its probability of winning is 1/2; the other two vertices each have winning probability 1/4.

The final outcome of the abacus for the tree in Figure 2 is shown in the augmented directed graph, Figure 5.

![Figure 5](image)

**Figure 5** Final outcome of the abacus on the random rooted tree.

Chips added: 212

The gray circular vertices are the internal vertices and the top vertex is the root. These vertices are labeled with the number of chips that are initially loaded into each vertex to satisfy the critical loading condition. The square white vertices are the absorbing vertices, one for each vertex in the tree. A total of 212 chips were added to the abacus after its initial critical loading, at which point the critical loading levels have been reached once more. This means that the root, whose chip count in its absorbing vertex is 91, has win probability 91/212.

A tree of this size is just about on the border of the sizes for which the abacus can reasonably be completed manually. There are programs that can implement the
abacus—this is how the outcome above was actually computed—but they are limited. Relatively simple trees require thousands of steps with this approach. For example, the stochastic abacus deposits nearly 28 million chips into the absorbing vertex of the root of the tree in Figure 6. This number will be defined below as the restoration number of the rooted tree.

![Figure 6](image_url) A slightly larger tree with high restoration number.

**Direct computation of the abacus**

Next we describe how results can be computed much more easily. The rooted tree with root \( r \) and subtrees \( T_1, T_2, \ldots T_m \) is denoted by

\[
\text{RootedTree}(r; T_1, T_2, \ldots T_m).
\]

**Definition.** The *restoration number* of a rooted tree is the number of times the root needs to fire to return the stochastic abacus to its critical loading position, denoted by \( R(T) \).

The restoration number is also the number of chips in the root’s absorbing vertex upon return to critical loading. The restoration number of the tree depicted in the abacus in Figure 4 is 91.

**Definition.** The *restoration function* of a rooted tree \( T \) is the function \( R_T \) on the vertex set of the tree such that \( R_T(v) \) is the number of chips in \( v \)’s absorbing vertex upon return to critical loading.

Note: The probability that vertex \( v \) wins Pass the Buck on a rooted tree \( T \) is

\[
\frac{R_T(v)}{\sum_{w \in V_T} R_T(w)}.
\]

**Definition.** The *period* of a rooted tree \( T \) is the restoration number of the rooted tree having \( T \) as its only subtree. The period of a tree \( T \) is denoted by \( P(T) \).

Note: If \( T' = \text{RootedTree}(v; T) \), then \( R_{T'}(v) = P(T) \).
**Theorem 1.** If $T$ has root $v$ and has $m$ subtrees with roots $(v_1, v_2, \ldots, v_m)$, then

$$P(T) = (m + 2)R_T(v) - \sum_{i=1}^{m} R_T(v_i).$$

(1)

**Proof.** If we create a tree with root $w$ having $T$ as its only subtree, the critical loading condition at $w$ occurs whenever $T$ is critically loaded, in which case $w$ can get sufficient chips without further firing. Therefore, we need to count how many chips $v$ needs. In order for $v$ to fire, it needs to receive $m + 2$ chips each time, where $m + 2$ is the outdegree of $v$ in the stochastic abacus. The number that it needs, $(m + 2)R_T(v)$, does not all come from $w$, however. This number is decreased by one every time any of the roots of the subtrees of $v$ fire, which accounts for the sum that is subtracted in equation (1).

The period of the tree depicted in Figure 5 is $(4 \cdot 91) - (26 + 35) = 303$. The significance of the period of a tree is that when several subtrees combine with a root, the restoration number of the new rooted tree is a function of the periods of its children.

**Theorem 2.** Let

$$T' = \text{RootedTree}(v; T_1, T_2, \ldots, T_m),$$

then

$$R_{T'}(v) = \text{lcm}(P(T_1), P(T_2), \ldots, P(T_m)).$$

(2)

and for each vertex $w$ in $T_k$, we have

$$R_{T'}(w) = \frac{R_{T'}(v)}{P(T_k)} R_{T_k}(w).$$

(3)

**Proof.** As the stochastic abacus is running, each subtree $T_i$ reaches its own critical loading condition after $P(T_i)$ root firings. Therefore, critical loading of all subtrees is first reached after the least common multiple of their periods, establishing equation 2. For each of the subtrees, the number of chips deposited in a period is multiplied by the number of periods that the subtree goes through, which establishes equation (3).

This lets us determine the restoration function of any rooted tree from the bottom up. We illustrate the technique with the tree $T$ in Figure 2. Let $\epsilon(v_i)$ be the trivial tree with a single vertex, $v_i$, as its root. We know that $R(\epsilon(v_i)) = 1$, and $P(\epsilon(v_i)) = 2$, for $i = 1, 2, 3$. Therefore, the level 2 tree with three trivial subtrees,

$$\tau_1 = \text{RootedTree}(v; \epsilon(v_1), \epsilon(v_2), \epsilon(v_3))$$

has restoration value $R(\tau_1) = 2$. The period of $\tau_1$ is

$$P(\tau_1) = (3 + 2)R(v_4) - 3R_{\tau_1}(\epsilon) = 7.$$ 

On the right side of the tree, we have the subtree

$$\tau_2 = \text{RootedTree}(v_7; \text{RootedTree}(v_6; \epsilon(v_5))).$$

We can determine the restoration function and period of this tree:

$$R_{\tau_2}(v_5) = 1, \quad R_{\tau_2}(v_6) = 2, \quad R_{\tau_2}(v_7) = 5, \quad \text{and} \quad P(\tau_2) = 13.$$
Finally, we can compute the restoration function of $\mathcal{T}$:

$$R_{\mathcal{T}}(v_8) = \text{lcm}(7, 13) = 91.$$  

We complete the computation of $R_{\mathcal{T}}$ by multiplying $R_{\tau_1}$ by 13 and $R_{\tau_2}$ by 7. The final result agrees with the actual implementation of the stochastic abacus that was displayed in Figure 5.

**Implementation of the direct calculation**

In order to implement the process described above, we use an array representation of rooted trees. We number the vertices in a tree with $n$ vertices with the positive integers from 1 to $n$. The structure of the tree is encapsulated in an array of $n$ integers, $T$.

In general, the entry $T[k]$ contains the parent of vertex $k$. The root of the tree has no parent and if $k$ is the root, and therefore $T[k] = \emptyset$. If we number the vertices in Figure 2 by the subscripts of the vertex names, then the tree would be represented by the array $\langle 4, 4, 4, 8, 6, 7, 8, 0 \rangle$.

The following Mathematica code will identify various parts of a rooted tree, assuming the structure we have described above:

\[
\langle 6 \rangle \equiv
\]

\[
\text{root}[T_] := \text{FirstPosition}[T, \emptyset] \quad /\quad \text{First}
\]

\[
\text{children}[T_, k_] := \text{Position}[T, k] \quad /\quad \text{Flatten}
\]

\[
\text{leafQ}[T_, j_] := \text{Not}[\text{MemberQ}[T, j]]
\]

\[
\text{descendants}[T_, j_] := \{\}\quad /\quad \text{leafQ}[T, j]
\]

\[
\text{descendants}[T_, j_] :=
\]

\[
\quad \text{Join}[\text{children}[T, j],
\quad \text{Join @@ Map}[\text{descendants}[T, #], \text{children}[T, j]]] /\quad \text{leafQ}[T, j]
\]

The following functions convert a tree in the form of undirected edges with designated root into the array we use in our implementation.

\[
\langle 7 \rangle \equiv
\]

\[
\text{treeArray[el_List, root_]} :=
\]

\[
\text{If}[\text{AcyclicGraphQ}[\text{Graph}[el]] \quad \&\quad \text{ConnectedGraphQ}[\text{Graph}[el]],
\]

\[
\quad \text{maketreeArray[el, root], \quad \text{`error`}]
\]

\[
\text{maketreeArray[el_List, root_]} :=
\]

\[
\quad \text{Module}[\{ta, n\}, \quad n = \text{Length}[el] + 1; \quad ta = \text{Table}[0, \{n\}];
\]

\[
\quad \text{Map}[\text{FindShortestPath}[el, \text{root}, \#], \quad \&\quad \text{Complement}[\text{Range}[n],
\]

\[
\quad \quad \{\text{root}\}] /\quad \text{Map}[\text{Partition}[\#, 2, 1], \quad \&\quad \text{Flatten}[\#, 1] \quad \&\quad \text{Union} /\quad \text{Map}[\text{ta}[\text{[[1]]}] = \text{[[2]]], \quad \&\quad \text{ta]}
\]

This next function computes the restoration function of a tree in the form of a list of undirected edges with designated root. The expression $r[T, k, j]$ represents the restoration function of the subtree within $T$ rooted at $k$ evaluated for the vertex $j$; and $p[T, k]$ is the period of the subtree of $T$ rooted at $k$. 
\[8\] \equiv 

\text{restoration}[\text{tree}, \text{root}] := 
\text{Module}[\{r, p, \text{ta}, n\}, 
\text{ta} = \text{treeArray}[\text{tree}, \text{root}]; 
\text{n} = \text{Length}[\text{tree}] + 1; \ r[\text{T}, k_, k_] := 1 /; \text{leafQ}[\text{T}, k]; 
\text{p}[\text{T}, k_] := 2 /; \text{leafQ}[\text{T}, k]; 
\text{r}[\text{T}, k_, k_] := 
\text{LCM} @@ \text{Map}[\text{p}[\text{T}, #]&, \text{children}[\text{T}, k]] /; \text{Not}[\text{leafQ}[\text{T}, k]]; 
\text{p}[\text{T}, k_] := 
\text{p}[\text{T}, k] = (2 + \text{Length}[\text{children}[\text{T}, k]]) \ r[\text{T}, k, k] - 
\text{Total}[\text{Map}[\text{r}[\text{T}, k, #]&, \text{children}[\text{T}, k]]]; 
\text{r}[\text{T}, k_, j_] := 
\text{r}[\text{T}, k, j] = 
\text{Module}[\{i\}, 
\quad i = (\text{Select}[\text{children}[\text{T}, k], 
\quad \text{MemberQ}[\text{Join}[\#, \text{descendants}[\text{T}, \#], j]]]) // \text{First}; 
\quad \text{r}[\text{T}, k, k] = \text{r}[\text{T}, i, j]/\text{p}[\text{T}, i]; 
\quad \text{Map}[\#, \text{r}[\text{ta}, \text{root}, \#]] & \text{Range}[\text{n}] \} ]

Here we test the code with the example of Figure 2 and see that it is consistent with the output of the abacus seen in Figure 4.

\[9\] \equiv 

test1 = \{ \{ \text{UndirectedEdge}[8, 4], \text{UndirectedEdge}[8, 7], \text{UndirectedEdge}[4, 1], \text{UndirectedEdge}[4, 2], \text{UndirectedEdge}[4, 3], \text{UndirectedEdge}[7, 6], \text{UndirectedEdge}[1, 8] \}; 

\text{restoration}[\text{test1}, 8] 
\{ \{1, 13\}, \{2, 13\}, \{3, 13\}, \{4, 26\}, \{5, 7\}, \{6, 14\}, \{7, 35\}, \{8, 91\} \}

The tree in Figure 6 is more complex, yet also not huge, but its restoration number is considerably larger than the previous example.

\[10\] \equiv 

test2 = \text{Map}[\text{UndirectedEdge}@@#&, 
\{ \{26, 22\}, \{25, 24\}, \{25, 23\}, \{25, 22\}, \{22, 15\}, \{21, 17\}, 
\{20, 19\}, \{20, 18\}, \{20, 17\}, \{17, 16\}, \{16, 15\}, \{15, 1\}, 
\{14, 10\}, \{13, 12\}, \{13, 11\}, \{13, 10\}, \{10, 3\}, \{9, 5\}, \{8, 7\}, 
\{8, 6\}, \{8, 5\}, \{5, 4\}, \{4, 3\}, \{3, 2\}, \{2, 1\} \} ]

\text{restoration}[\text{test2}, 1] 
\{ \{1, 27783522\}, \{2, 10297681\}, \{3, 3109521\}, \{4, 1158449\}, \{5, 365826\}, 
\{6, 60971\}, \{7, 60971\}, \{8, 121942\}, \{9, 182913\}, \{10, 981954\}, 
\}
Extension to some non-trees

Although we have outlined a procedure for implementing the abacus on rooted trees, these results can be extended to some non-trees using so-called inconsequential edges, as introduced by Torrence [7]. If there are two isomorphic subtrees of a vertex in the tree, then an edge can be added to connect corresponding vertices of the subtrees. The restoration function of the new graph, which is not a tree, is equal to the restoration function of the tree, letting us compute probabilities for Pass the Buck on the new graph. This follows from the fact that since the two subtrees are isomorphic, the abacus on each subtree is mirrored in the corresponding subtree. For every chip that moves in one direction along the added edge, another chip moves in the other direction, so that the two movements offset one another. Thus, the added edge is deemed inconsequential.

A situation where several inconsequential edges can be added would arise if we created the tree $T' = \text{RootedTree}(r; T, T)$, where $T$ is the rooted tree in Figure 2. Note that since $P(T) = 303$, $R_{T'}(r) = 303$. The restoration function $R_{T'}$ is unchanged if we add an edge between the two copies of $v_k$, with $4 \leq k \leq 8$ or between any vertex in the set $\{v_1, v_2, v_3\}$ of the first subtree with a vertex in the corresponding set in the other subtree.

REFERENCES

[1] Bjöörn, A., Lovasz, L., Shor, P. (1991). Chip-firing games on graphs. Eur. J. Combin. 12(4): 283–291. doi.org/10.1016/s0195-6698(13)80111-4
[2] Engel, A. (1976). Why does the probabilistic abacus work? Educ. Stud. Math. 7(1–2): 59–69. doi.org/10.1007/BF00144359
[3] Levasseur, K. (2022). Pass the Buck on a complete binary tree. Math. Mag. 95(4): 349–353. doi.org/10.1080/0025570X.2022.2094165.
[4] Propp, J. (2018). Prof. Engel’s marvelously improbable machines. Math Horizons. 26(2): 5–9. doi.org/10.1080/10724117.2018.1518840.
[5] Snell, J. L. The Engel algorithm for absorbing Markov chains. https://arxiv.org/abs/0904.1413v1. Last accessed: January 2023.
[6] Torrence, B., Torrence, R. (2016). Fibonacci, Lucas, and a game of chance. Math. Mag. 89(5): 342–351. doi.org/10.4169/math.mag.89.5.342
[7] Torrence, B. (2019). Passing the buck and firing Fibonacci: Adventures with the Stochastic Abacus. Amer. Math. Monthly. 126(5): 387–399. doi.org/10.1080/0029890/2019/1577089

Summary. The Stochastic Abacus can be employed to compute winning probabilities for each vertex of a rooted tree in the game “Pass the Buck,” with the starting vertex being the root of the tree. For all but the simplest trees, the abacus cannot really be implemented due to the large number of steps needed for completion. In this paper, a technique for anticipating the outcome is introduced.

KENNETH LEVASSEUR (MR Author ID: 201640) is a professor at the University of Massachusetts, Lowell, where he has been on the faculty since 1980. He has a Ph.D. in mathematics from the University of Rhode Island, where he studied under Jim Lewis. His main interests are in discrete mathematics and problem solving. In his spare time he enjoys golf, tennis, card collecting, and playing with his granddaughters and grandson.