DISCLAIMER: This is merely an improvement upon PageRank and does not claim to be sybil-resistant, solve double spending or forking problems, or calculates a perfect reliable trustworthiness (but neither does PageRank).

Abstract

The proposed algorithm combines block discovery and PageRank calculation, and makes it an integral part of executing transactions. Its implementation only requires a single field to be added to the trustchain blocks.

Most of the time it acquires the same result as the PageRank algorithm would converge to if ran for an infinite number of random walks[1].

The algorithm is truly distributed and only requires interaction between the requesting and supplying party. It furthermore only requires interaction when a party tries to initiate a transaction with another party.

The algorithm comes at the cost of an average runtime-complexity of O(E[degree]) and an average space-complexity of O(E[degree]^2) [2].

The algorithm can be easily adapted to be equivalent to the results of the PageRank algorithm with four-hop random walks.

Introduction

This algorithm is an adaptation of the PageRank algorithm, it yields the same results and exhibits the same properties but can be calculated in a distributed manner. It deals with misreports by only using signed blocks.

To implement this algorithm, addition of a ‘pair-wise running total’ field is needed in the blocks. This field holds the running total of all transactions between the two parties creating this block. At first this might seem a counter-intuitive approach of reducing the memory footprint by storing more data, but this enables any arbitrary edge in a graph to be represented with a single block regardless of the total chain length of both parties.

The algorithm is run every time a party wants to acquire resources from another party. When a party receives this request, it can be from a known party or an unknown party. We will first explain the Initial Ranking algorithm for unknown parties and later extend the algorithm to handle transactions with known parties by defining the Incremental Ranking Algorithm.

Initial Ranking Algorithm

Every party naturally already knows his one-hop neighbors as this is stored in his personal

[1] The exact conditions required for which the score is incorrect are described in the section ‘Accuracy’

[2] Please be aware that E[degree] is the average number of unique one-hop neighbors, and not the total amount of blocks you created with your neighbors. At the moment of writing we had acces to 4.000.000 blocks. In the graph created with these blocks the nodes had an average degree of 41 with a 95th percentile being 75 or lower.

TrustChain. As stated earlier every block contains a pair-wise running total, so a party can perform O(1) lookups to determine the weight of an edge in his one-hop neighborhood.

Now assume Bob wants some data from Alice. To do so he will send Alice a request. On reception of this request Alice will demand the complete TrustChain of Bob. By supplying Alice with his TrustChain Bob basically transfers his one-hop neighborhood to Alice.

Alice will now check the chain and verify its completeness and correctness. If the chain is incomplete or incorrect Alice can terminate the request, since Bob (who she never met before) starts his first interaction by supplying incorrect or incomplete data.

After the chain verification Alice can discard the majority of the blocks: The pair-wise running total already describes a complete edge in a graph. This means that for every party in Bob's one-hop neighborhood Alice only has to store a single block (On average this will be O(E[degree] blocks) .

An important realization is that since this algorithm was also executed for all of Alice's one-hop neighbors (one only becomes a one-hop neighbor after transacting), Alice has a complete overview of her two-hop neighborhood.

The next step for Alice is to find a match with any of Bob's one-hop neighbors in Alice's two-hop neighborhood. A match amongst her one-hop neighbors means there is a two-hop path to Bob, and a match in here two-hop neighborhood reveals a three-hop path.

Alice now knows all the paths of three hops and shorter that lead to Bob, knowing this it is very easy to calculate the probability of taking that path. In the original PageRank algorithm random walks have a probability of being terminated at every hop. To account for the termination probability in our algorithm we use the fact that the decision to terminate is a Bernoulli trial. To do so every weight encountered on the path is multiplied by $(1-p)^n$, where n is the number of hops taken and p the termination probability.

Summing the probabilities of all paths gives Alice the PageRank value of bob. In almost all cases this result will be equal to that of the PageRank algorithm, had she run it for an infinite number of three-hop random walks.

If no path exists the probability of ending up at Bob is zero, for both this algorithm and the original PageRank algorithm the result will be a score of zero.

Incremental Ranking Algorithm

In the previous section we only considered interactions with parties that were unknown to us. But there exists a possibility that we have interacted with the requesting node before.

On the reception of a request, Alice should check if the requesting party is in our one-hop or two-hop neighborhood. If the party only exists in the two-hop neighborhood it will be considered as an unknown party, and the Initial Ranking Algorithm will be used.

If Alice receives a request from Bob, and Bob is a one-hop neighbor, Alice searches for the last block Alice and Bob made together in her chain. Alice can then request all the blocks in Bob's chain that have been created after their last commonly created block. This results in significantly reduced communication overhead.

After receiving the new blocks Alice updates the existing parties and adds any new parties to her two-hop neighborhood. She then only has to recalculate the paths that have been affected by the blocks to generate the new PageRank score. This can be done efficiently since the difference in Bob's PageRank score is the sum of all differences between the old paths and the new paths.

Extending to four-hop walks
The algorithm can be easily extended to obtain the results equivalent to four-hop random walks. To do so the requesting party should not only send it's one-hop neighbors (i.e. his TrustChain) but also a single block per two-hop neighborhood. In this way it is possible two calculate paths through parties in the two two-hop neighborhoods, resulting in four-hop paths.

This has the implication that either the source node should store these extra blocks (resulting in a space-complexity of (E[degree]^3)) or the requesting party should send his two-hop neighborhood in every subsequent transaction increasing the communication overhead.

Either way, this operation is more expensive than three-hop lookups, therefore it might be an option to only perform these extended walks when the result of the three-hop lookups is below a certain threshold.

Misreport Resilience
In case of a three-hop random walk all the data required by Alice to calculate Bob's ranking is signed and originates from a chain Alice has personally inspected. Therefore she can rely on the data being correct (correct in the sense that the counter party of that block can't deny the report as it contains their signatures). Hence the three-hop neighborhood implementations can be considered as misreport resilient.

In case of the four-hop random walk this argument doesn't hold. The data about Bob's two-hop neighborhood is passed along as a single block per party instead of a complete chain, so Bob might have selectively presented Alice blocks that are beneficial for his ranking.

Such a selective block presentation attack has a limited effectiveness, for which we shall calculate the upper-bound in case of a single four-hop path. Since information on Alice her one- and two-hop neighbors and on Bob's one-hop neighbors all originate from a complete chain verified by Alice, Bob can only selectively present blocks that would be three-hops away from Alice. Just as in PageRank the termination probability is implemented, but here it is done using the expected value of the walk not being terminated after n-hops. The probability of reaching the third hop is $(1-p)^2$ where p is the termination probability.

This means that the error that can be induced in a single four-hop path by is limited to the difference between the best-case block and the worst-case block scaled by a factor of $(1-p)^2$.

Handling of stale blocks
After finding a match in each other's neighborhoods there exists a chance that one of the two parties holds a block that is older than the other one (which can be derived from the sequence numbers). If this stale block is found in a two-hop path it will not affect the result, as the edge between you and your one-hop neighbors can only be changed by transactions between you and that specific one-hop neighbor.

If this stale block is found in a three-hop path it may affect the result. The impact of a stale block completely depends on the amount (and total size) of transactions that has passed since that block.

One way of minimizing the impact of stale blocks is to regularly ask your one-hop neighbors for the latest blocks since the last transaction. This results in a more up-to-date view of your one- and two-hop neighborhood.

Another simple but effective mechanism is that not only the requesting, but also the receiving party sends his chain. This does increase the communication of the algorithm but leaves the network better informed.

Regardless of how the stale blocks wil be handled, both parties will always be informed if their current view of the graph is not entirely up to date.

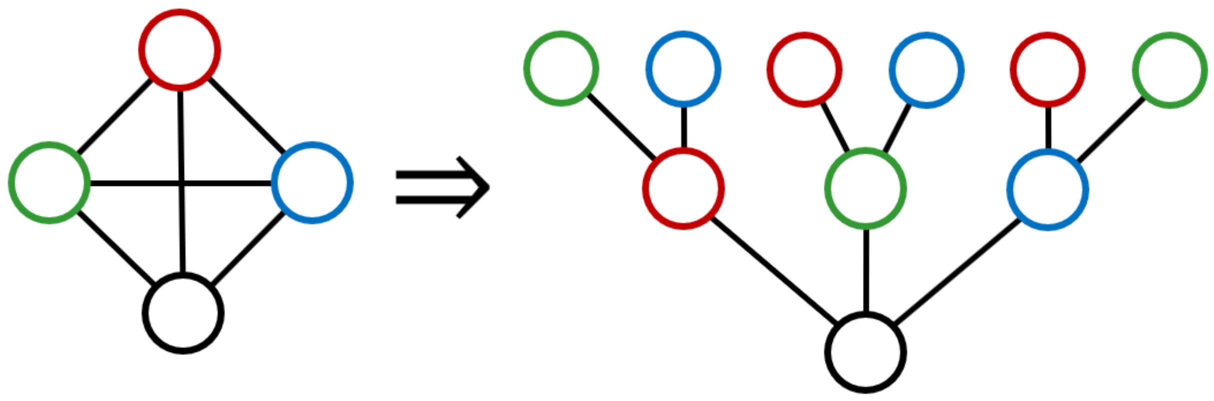
Accuracy
Overall this algorithm will yield a more accurate result, since it can directly calculate the values PageRank algorithm will approach when ran for infinite random walks. Considering the algorithm will only perform finite number of random walks, there will always be an error.

However, there exists a very specific case where the algorithm will have a lower score than the original implementation. This is due to the fact that random walks are expected to be of length three, while longer walks actually have a non-zero probability. In almost all cases this has no impact, except for the case where Bob can visit himself through a cycle, and all of the nodes in the cycle are not within Alice's three-hop neighborhood.

While there isn't yet an exact number of the likelihood of this specific case, it seems likely that the improved accuracy on all the other cases will compensate for the error induced in this very specific case. But no hard numbers can be given as we haven't yet verified the probability of occurrence of such cycle or the impact on the score.

Implementation optimizations
One of the advantages this algorithm has is that both the 'running total' (the sum of all outgoing edges) and the 'pair-wise running total' (the weight of the edge of interest) is stored in every block. This means the latest outgoing probability can be calculated simply dividing the two values from a single block instead of iterating over all outgoing edges.

A second optimization lies on how data about the neighborhoods are stored. As every block describes an edge, receiving someone's chain is equivalent to receiving someone's edges. While it seems intuitive to think of the transactions as a graph, it is more efficient to think of it as a tree. Doing so eliminates cycles and removes the need for a routing algorithm to find paths in the graph. If you need to find all the paths to a node, you simply start at every occurrence in the three and traverse to its parent until you reach yourself.



When looking at the figure above the left graph might look more memory efficient to store, as it only has 6 edges (which requires 6 blocks to be stored), where the right tree has 9 blocks.

However, seeing that the pair-wise running total is symmetrical, an edge in the right graph traversing from red to green is described by the exact same block as the edge traversing from green to red. Using references to blocks instead of multiple copies of blocks we can describe any graph as a tree without having to store any additional blocks.