

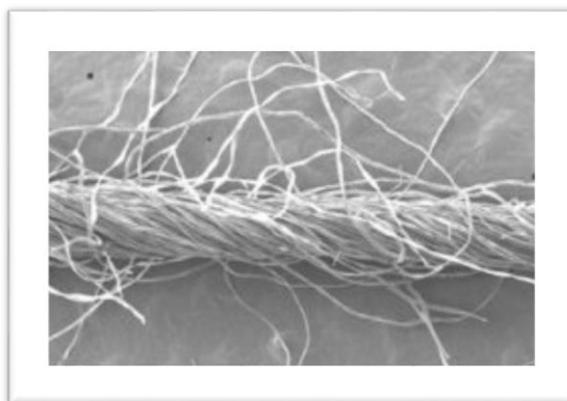
**DagCoin Draft**  
**September 11, 2015**  
Sergio Demian Lerner

**Abstract**

DagCoin is a cryptocurrency design that attempts to be highly decentralized by merging the concepts of transactions and blocks and making each user that transact a miner. Each transaction carries a proof-of-work and references one or more previous transactions. The resulting authenticated data structure is a Direct Acyclic Graph (DAG) of transactions where each transaction “confirms” one or more previous transactions. The confirmation security of a transaction is measured in accumulated amount of proof-of-work referencing the transaction. In this paper we present the DagCoin design, solve the double-spend problem and show several optimizations to aid for an efficient implementation.

DagCoin is a cryptocurrency design that merges the concepts of transactions and blocks and making each user a miner. Each transaction carries a proof-of-work and references one or more previous transactions. The resulting authenticated data structure is a Direct Acyclic Graph (DAG) of transactions where each transaction “confirms” one or more previous transactions. The confirmation security of a transaction is measured in accumulated amount of proof-of-work referencing the transaction. This structure is better suited for a cryptocurrency without subsidy (such as a side-chain), since the cost of reversal of a transaction can be easily measured, where in merged-mining the reversal cost depends on the good will of the non-merged hashing power.

One of the problems with the DAG approach is how to limit the maximum cut of the generated DAG or, in other words, how to prevent all new transactions from referencing the same set of parent transactions, and degenerating the DAG into a star graph. The DAG must not increase in “width”, and it must “look” more like a yarn under microscope. I will call this structure a DAG-chain.



A DAG-chain can be informally defined as DAG that:

- After taking all border (non-parent) nodes  $k$  times, it becomes a chain
- The resulting chain length is proportional to the original node count by a factor close to  $2k$ .
- If the DAG has more than  $2k$  nodes, you can cut it in two separate DAGs, and the same properties hold for each half (each half having a factor  $k$  which is close to the original  $k$  factor).

To be able to create a DAG-chain the protocol must prevent users from choosing old transactions to extend the DAG. Merging branches should be incentivized, but not too much such that users merge the same branches over and over. The problem of spam is of less importance, as no transaction can get a “free ride” in a block. We show that the election of an adequate data structure allows the DAG-

chain to be formed, but it requires us to change how we think about double-spends.

The premises used to design the DagCoin cryptocurrency are the following:

*PREMISE: The cryptocurrency network benefits from creating a DAG-chain growing as “thin” (low  $k$ ) as possible.*

In other words, having the average maximal cut as low as possible. Referencing many previous transactions (high out degree) can make the DAG thinner only if the following transactions reference the transaction with high out degree, but are themselves of low out degree. The DAG requires high out degree some times, but low out degree another times.

DagCoin tries to fulfill that premise, using an incentive structure such that:

- There is a benefit for users to reference as many previous transactions as possible
- Referencing many previous transactions is incentivized only when there are many previous transactions unreferenced.
- There is no competition between users to reference a previous transaction.

### **Safely accepting Double-spends in the DAG-chain**

In Bitcoin, a transaction in a valid block-chain can never be a double-spend, as double-spending violates a protocol rule. DagCoin allows two conflicting transactions to be included in the DAG-chain as long as the second does not reference the first (over one or more hops). We assign each transaction a confirmation score. If two conflicting transactions appear, as more transactions are added to the DAG-chain, the number of confirmations of one of the two will increase, but the other will not. Each transaction adds one unit of confirmation. The score of a node without children is zero. The score of a referenced transaction is the sum of all transactions that recursively reference it (including double-spends). Whenever a transaction is added, it modifies the scores of all transactions recursively referenced by it. Whenever a transaction references a list of previous transactions, if there are two conflicting transactions, then the one with highest score prevails. If both have the same score, then the order of referencing establishes preferences over the conflicting transactions, such that the first transaction gets its score increased but any following double-spend will not.

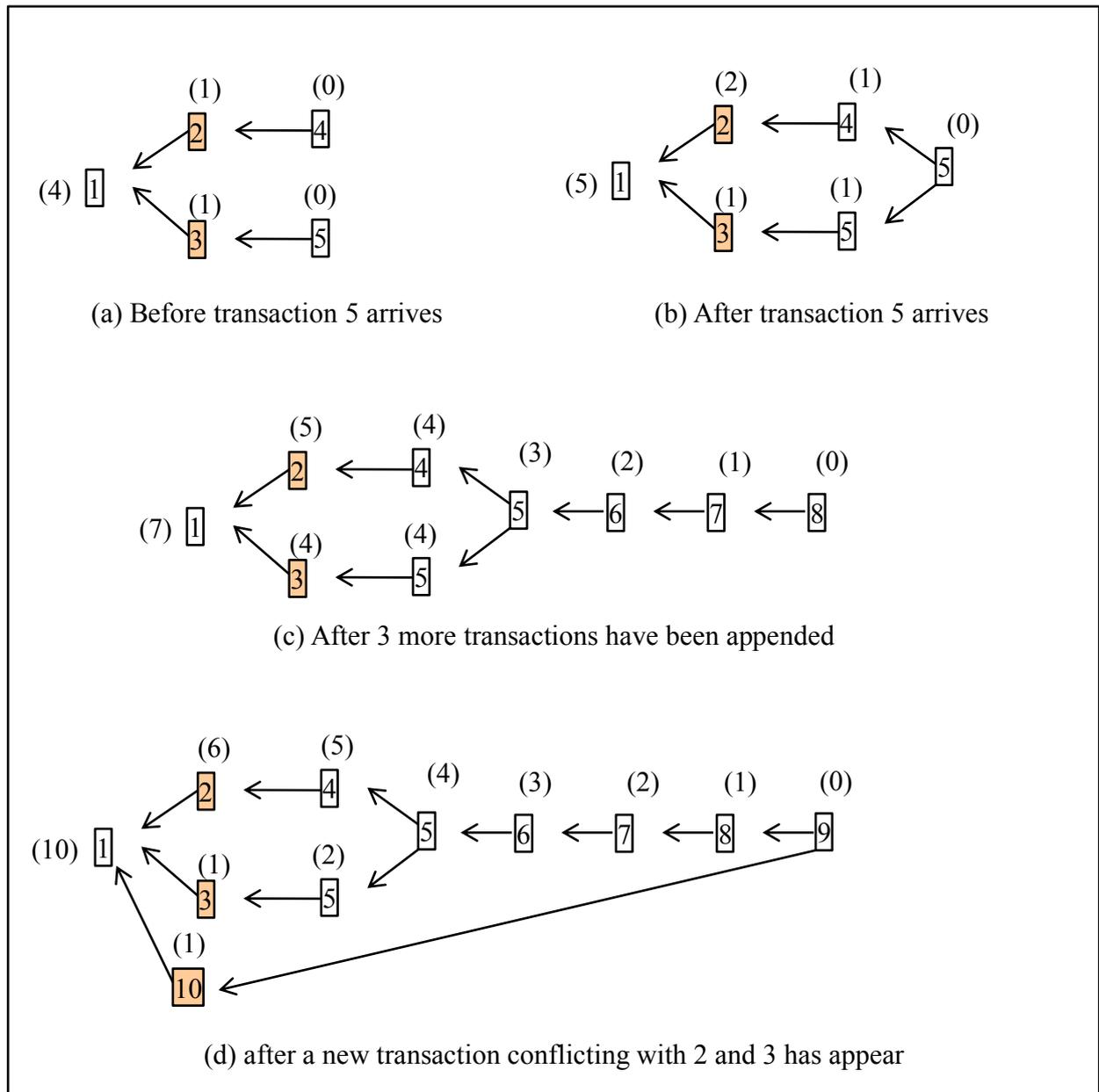


Figure 1

Figure 1 shows the DAG before a join transaction arrives and afterward. Transactions 2 and 3 (in orange) are conflicting. The confirmation score is in brackets. We can see that even both transaction 2 has transaction 3 have a non-zero confirmation score, only one of them will increase over time. Honest nodes will never extend a transaction which is already referenced, so an attacker that wants to replace transaction 2 by transaction 10 must invest in proof-of-work at least the difference between the confirmation scores. This establishes a very precise bound on the double-spend security.

### Preventing too many transactions merging too many transactions

The core idea proposed is that each transaction commits to an authenticated forest of previous unreference transactions. To do so, it includes the value  $C(N)$ , where  $C(i) = \text{Commit}(C(i-1) \parallel T(i))$ , where

$T(i)$  is the hash of a transaction parent and  $C(0)$  is the empty string. These are simple recursive commitments so that  $C(N)$  allows the payer to reveal any number of parent hashes between 1 to  $N$ . The important decision is how many parents the commitment should reveal. Using the transaction as a header, the payer tries to find a proof of work with certain base difficulty (more on this base difficulty later). If the obtained a proof-of-work whose difficulty is  $2^k$  times harder than the base difficulty, it will reveal and reference the first  $(k+1)$  nodes of the list. Half of the times a transaction will have a single parent, so only the first node  $T(N)$  will be revealed, by providing the complementary hash chain head  $(C(N-1))$ . One fourth of the times, two transactions will be referenced, by providing the hashes  $T(N)$ ,  $T(N-1)$  and  $C(N-2)$ .

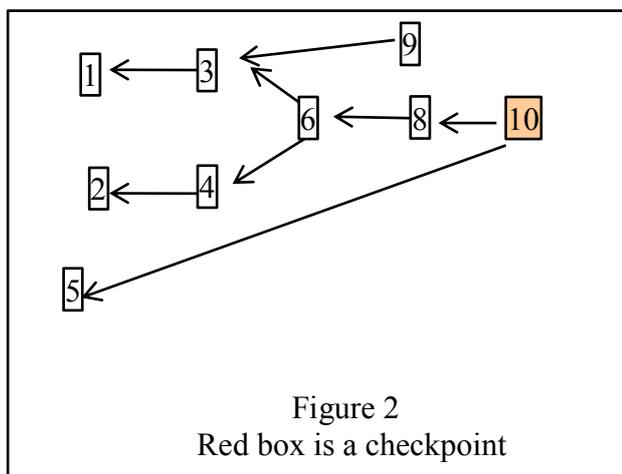
This system provides a logarithm distribution in the amount of parents, with an average of 2. Also this method cannot be gamed, since referring more parents has a PoW cost.

There should be an incentive to include as many references as possible in the authenticated branch. This can be achieved by several methods:

1. Invalidating a transaction that has less references than what the PoW requires.
2. Incrementing the score of a transaction that has more revealed references. For instance, a transaction having  $K$  revealed references could add a fractional score of  $(K-1)/K$  to the transaction score.

### Preventing Unbounded Cascade Updates to Confirmation Scores

Suppose that for each transaction we save an integer score that we update for each new transaction that references it directly or recursively. It is evident that the proposed data structure requires updating almost all previous confirmation scores each time a transaction is added. To reduce the workload, we use pointers and checkpoints. At a certain frequency the software chooses a transaction that references a high number of parent nodes. Figure 2 show how a checkpoint is found.



Of course, not every past transaction could be reachable, as users may decide to never reference certain published transaction. However, the parent selection, with average out-degree 2, and low network latency, can guarantee that there will be frequent checkpoints referencing almost all previous transactions.

After a checkpoint is found, the software updates all nodes reachable from the checkpoint with a forward pointer to this checkpoint. A checkpoint has its own score counter, initially set to zero. When the update algorithm reaches a checkpoint node, it increments the counter and stops propagating backward. The score of a transaction is computed as the last stored score in the transaction plus the score of the pointed checkpoint. Checkpoints are considered as nodes on the DAG, so the same checkpoint finding algorithm can make checkpoints that refer to other forward checkpoints. After several continuous checkpoints, a checkpoints of a higher level is created referencing previous checkpoints, forming a skip-list. Using the skip-list it is possible to compute the score of a transaction in  $O(\log N)$  where  $N$  is the number of transactions after it. Also, after a certain score has been reached, the wallet may decide not to update it anymore and consider it immutable, as in Bitcoin checkpoints.

### Periodic re-computing to reduce computation load

Even using checkpoints, computation load can be high. When a wallet detects a transaction whose destination address is owned, it will start tracking it to find out how deep it is confirmed. But computing the confirmation score using every new transaction that arrives is expensive. To reduce the load, the wallet can re-compute the score after a certain amount of accumulated proof-of-work has been received, creating arbitrary “blocks” of transactions. Each block is then processed separately to find all the parent transactions “inputs” of the block, and a score is added to each input. It’s important not to confuse the block inputs with Bitcoin’s UTXOs, DagCoin block inputs are not related with spending and represent block parent hashes (instead of a single patent hash). The number of inputs will depend on the network latency, but will be generally low and independent of the block size. For example, for a network with 10 tps, and 1 second of propagation latency, the block input set cardinality should be around 10. Then the set of inputs is processed. Figure 3 show a block and how the input set is constructed (not necessarily in the same way) by each wallet software. Every input has an accumulated score which is propagated to previous nodes.

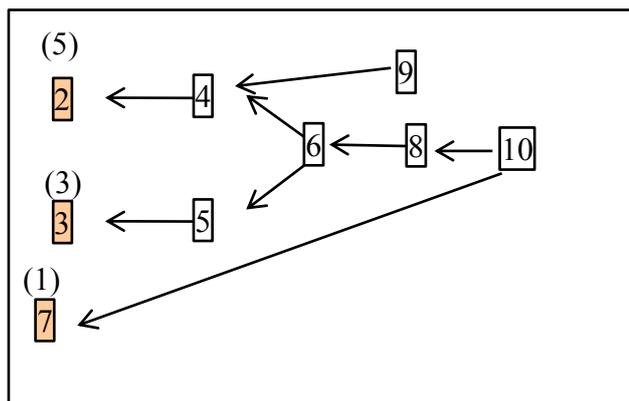


Figure 3  
Red boxes are the input set of the block

For instance, and to provide a comparison to Bitcoin, the wallet may consider 10K units of transaction PoW as equivalent 1 “block confirmation” and so pack 10K transactions into a block and re-compute the score every 10K transactions received. A better approach is to construct a block every  $N$  seconds, independent on the number of transactions in it. Note that if there are no transactions being performed after the monitored one, then the confirmation score does not change. This is a direct consequence of the nonexistence of a subsidy and Bitcoin will face the same problem if the price does increase constantly.

## **Targeting a fixed transactions/rate vs no maximum rate**

As there are no free-rides for transactions, the transaction/rate is limited by existent deployed computing power and electricity cost. By time-stamping every transaction, one could dynamically adapt the difficulty of the proof-of-work to achieve more fixed rate. But if the difficulty of a transactions depends on the difficulty of the parent transactions, then there may be incentives to choose old parent transactions instead of new ones to reduce the PoW required, if the current rate is over the fixed rate. Just to be sure Moore's law does permit spamming in the future, one could embed a re-targeting rule such that every 18 months the difficulty is doubled. It seems preferable that the last M transactions (such as M=10K) of a certain transaction vote on an increase or decrease of the difficulty of the following transactions (with small step changes). Then users could vote more freely on how the network should work without having any immediate benefit to bias voting. This is a similar problem as the current Bitcoin block-chain increase problem: only miners can vote, because user votes are prone to Sybil attacks. In DagCoin, every user can vote, as long as it transact.

## **Conclusion**

We've presented a new cryptocurrency design based on a DAG structure where there are no fixed blocks and where each transaction carries its own proof of work. Also we've presented two optimizations that allow storing and dynamically updating the DAG-chain consuming low CPU resources. It must be noted however that the proposed DAG-coin cannot verify new transactions using only a subset of the block-chain, such as Bitcoin's UTXO set. However, by storing the most recent transactions in a fast cache, and by using checkpoints where such that older transactions cannot be references, the system can be made as fast as Bitcoin, or faster.