Alan Viollier

CS 168 Final Project: Implementing Bitcoin Features to SpartanGold

Introduction

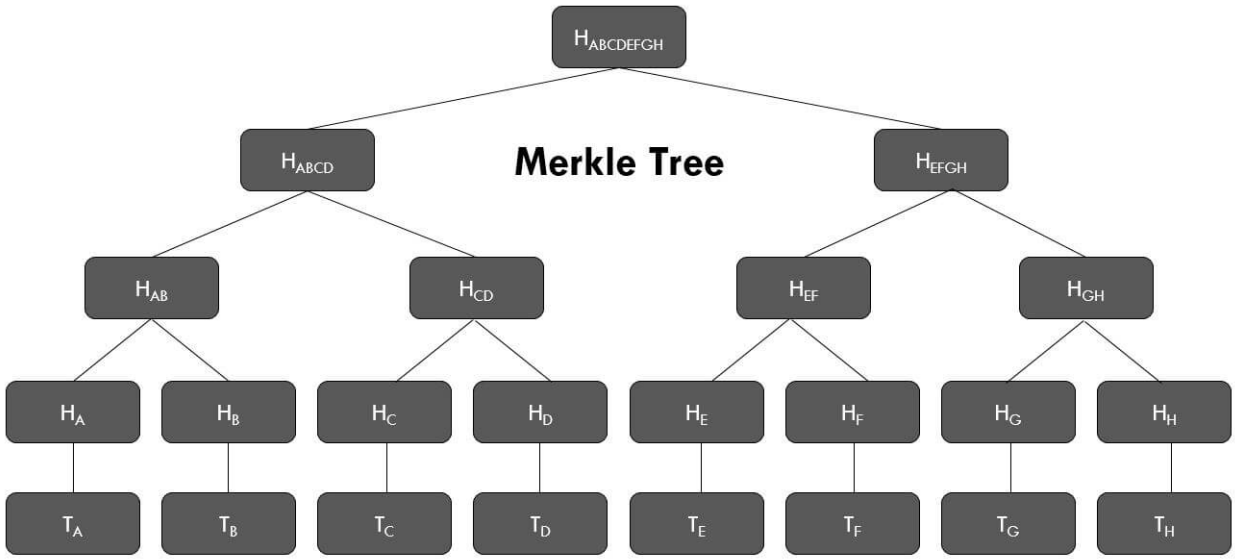
The purpose of this final project is to implement a significant functionality related to blockchains. This implementation could mean anything from adding features to an existing cryptocurrency, upgrading an already implemented feature, and anything else as long as it's related to blockchains. This project aims to use what we learned in class while also learning more about blockchains and pursuing our interests in this topic. In this project, I decided to implement Bitcoin features to SpartanGold. Bitcoin is the most popular cryptocurrency for its well-planned and robust components that have kept it going strong for many years with the trust people put into those features. SpartanGold, on the other hand, is just a sample cryptocurrency with more basic features used for learning. Let's look more in-depth at what a couple of Bitcoin's advanced features are compared to SpartanGold's basic elements.

Bitcoin stores transactions in a Merkle tree, a binary tree of transactions, while SpartanGold stores transactions in a javascript Map. Bitcoin has its limited scripting language for locking and unlocking transactions, while SpartanGold does not have any language. Bitcoin uses a UTXO-based model for storing transaction outputs, while SpartanGold only uses an Account based model. Bitcoin has

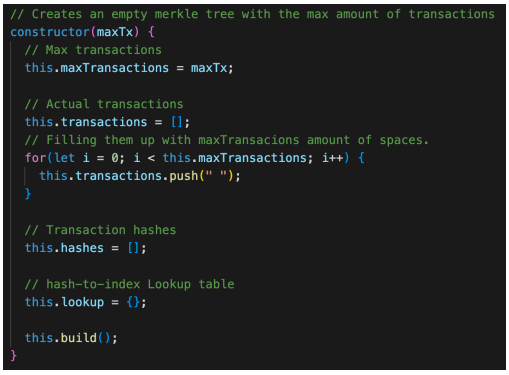
a proof-of-work that adjusts over time, while SpartanGold uses a fixed proof-of-work. Finally, Bitcoin uses a fixed block size while SpartanGold has no block size limit. Of many, these are the significant differences in features between Bitcoin and SpartanGold. In this project, I will be explaining how I implemented usage of the Merkle tree, fixed block size, and a proof-of-work adjusting over time into SpartanGold.

Merkle Tree

A Merkle Tree is a binary tree used in Bitcoin to store transactions. It does this by having all the leaf nodes be the hashes of all the transactions. Then, all the inner nodes are hashes of their children. This tree of hashes gives you a Merkle root stored in Bitcoin's blocks rather than transactions, making Bitcoin blocks a lot smaller in size. Validators can then verify transactions by recreating the hashes to the root and ensuring it's the same as the Merkle root. Bitcoin implemented this feature because it only takes log(n) hashes to verify a transaction, minimal data is needed to transmit the block over the network, and old transactions are pruned from the Merkle tree freeing more data over time.



To implement the Merkle tree feature into SpartanGold, I first created a MerkleTree class. When a new Merkle tree object is created, it takes in a transaction amount to build itself according to how many transactions it should be able to hold. Then, it makes an empty Merkle tree ready to store transactions. Essential functions of this class are the build and addTx functions. The build function is called on Merkle tree creation and whenever a transaction is added so that it re-hashes and rebuilds the whole tree. This rebuilding is so that the Merkle Tree is continuously updated to its most current version. Then we have addTx, which adds a transaction to the next open spot in the Merkle tree. The rest of the functions either help to get information, help build the tree or display the tree. Finally, I had to implement this class into the block class by replacing the Map with a Merkle tree and replacing Map functions with my Merkle tree class functions within the block, blockchain, client, and miner classes. This SpartanGold implementation now has a Merkle tree feature like Bitcoin.



```
// Creates an empty merkle tree with the max amount of transactions
constructor(maxTx) {
  // Max transactions
  this.maxTransactions = maxTx;

  // Actual transactions
  this.transactions = [];
  // Filling them up with maxTransacions amount of spaces.
  for(let i = 0; i < this.maxTransactions; i++) {
    this.transactions.push(" ");
  }

  // Transaction hashes
  this.hashes = [];

  // hash-to-index Lookup table
  this.lookup = {};

  this.build();
}
```

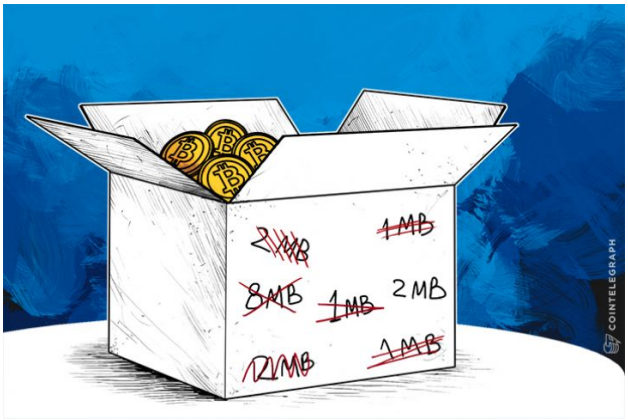
```
// Build out the MerkleTree with current transactions.
build() {
  // Merkle tree size.
  let numBalancedTree = this.constructor.calculateSize(this.maxTransactions);
  // First transaction position
  let firstTransaction = Math.floor(numBalancedTree / 2);
  // Adds the hashes for the transactions only
  for (let i=firstTransaction; i<numBalancedTree; i++) {
    let v = this.transactions[i-firstTransaction];
    let h;
    if(v == " ") {
      h = utils.hash(v);
    }
    else {
      h = utils.hash(v.getid());
    }
    this.hashes[i] = h;
    this.lookup[h] = i;
  }

  // Completing inner nodes of Merkle tree
  for (let i=firstTransaction+1; i<this.hashes.length; i+=2) {
    this.constructor.hashToRoot(this.hashes, i);
  }
}
```

```
// Adds a transaction to the merkle tree and rebuilds it.
addTx(tx) {
  // Check to see if it already full.
  if(this.isFull()) {
    console.log("This block is full!");
    return;
  }
  // Finds the next empty space and adds it.
  for(let i = 0; i<this.maxTransactions; i++) {
    if(this.transactions[i] == " ") {
      this.transactions[i] = tx;
      break;
    }
  }
  // Rebuild.
  this.build();
}
```

Fixed Block Size

Fixed Block size is a feature Bitcoin uses in which the block size is limited to 1 megabyte. This fixed size means only a certain amount of transactions can fit on a single block. Fixed block size is speculated to avoid overloading blocks with useless or malicious mass amounts of transactions. Miners are also encouraged to accept standard but not non-standard transactions as non-standard may take up extra space. Therefore, miners could use non-standard transaction space to collect fees from more standard transactions.



I implemented Fixed block size from Bitcoin into SpartanGold by using my last feature, the Merkle tree. I made my Merkle tree class a fixed Merkle tree, which means it can only be a specific size when created. Merkle tree can be full, and when adding transactions, it's filling up pre-made slots. When a block's Merkle tree is full, the block rejects the transaction. Limiting the size of the block isn't the only thing that needs to be implemented in this case; miners also need to be able to store transactions for the subsequent blocks if the current block is full.

```
addTransaction(tx, client) {
  if (this.txMerkleTree.has(tx)) {
    if (client) client.log(`Duplicate transaction ${tx.id}.`);
    return false;
  } else if (tx.sig === undefined) {
    if (client) client.log(`Unsigned transaction ${tx.id}.`);
    return false;
  } else if (!tx.validSignature()) {
    if (client) client.log(`Invalid signature for transaction ${tx.id}.`);
    return false;
  } else if (!tx.sufficientFunds(this)) {
    if (client) client.log(`Insufficient gold for transaction ${tx.id}.`);
    return false;
  } else if (this.txMerkleTree.isFull()) {
    if (client) client.log(`This Block is already full!.`);
    return false;
  }
```

```
postTransaction(outputs, fee=Blockchain.DEFAULT_TX_FEE) {
  // We calculate the total value of gold needed.

  let totalPaymentsNoFee = outputs.reduce((acc, {amount}) => acc + amount, 0);
  let realFee = totalPaymentsNoFee/10;
  let totalPayments = totalPaymentsNoFee + realFee
```

Miners also need to act greedy and add transactions to the current block that will make them the most money. So first, I made fees for each transaction in SpartanGold 10% of the transaction amount rather than a constant in the client class. This is not a good percentage or implementation for a cryptocurrency, but it helped me demonstrate miner greediness. Then, I made it so that every time a miner fills up a block, they add them by the highest fees and keep the rest of the transactions for the next block. I did this by putting all the transactions in an array and sorting them by their fee amount. I also made the miner check if the block's Merkle tree is full before adding every transaction; if it is full, the rest of the transactions were stored again. This SpartanGold implementation has a fixed block size feature and greedy miners like Bitcoin.

```
txSet.forEach((tx) => this.transactions.add(tx));

// Sort transactions by fee.
let sortable = []
// Get everything from transactions into array
for(let item of this.transactions) {
  sortable.push([item, item.fee]);
}
// sort the array
sortable.sort(function(a,b) {
  return b[1] - a[1];
});
// remove the fee column
let sorted = [];
for(let item of sortable) {
  sorted.push(item[0]);
}
// Back into transactions
this.transactions = new Set(sorted);

// For keeping the remaining transactions.
let newTransactions = new Set();
// Add queued-up transactions to block.
this.transactions.forEach((tx) => {
  // If not full add it to the block.
  if(!this.currentBlock.txMerkleTree.isFull()) {
    this.currentBlock.addTransaction(tx, this);
  }
  else {
    //otherwise save it for later.
    newTransactions.add(tx);
  }
});
// Add the transactions that couldn't make that block.
this.transactions = newTransactions;
```

Proof of Work Adjustment

Proof-of-work adjustment is a Bitcoin feature where blocks are mined about every 10 minutes no matter how much mining power there is. Bitcoin does this by getting the timestamps from previous blocks and adjusting the mining difficulty based on the time between recent blocks. If it's taking longer than 10 minutes, Bitcoin will make it easier to mine a block. Otherwise, if blocks are mined in under 10 minutes, Bitcoin will make it harder. This is why bitcoin is getting harder and harder to mine because more people are mining and better equipment mining over time. Implementing a feature like this is so that there is a steady flow of bitcoin that won't go rampant as more people mine. If blocks are mined faster and faster, then the value of bitcoin would plunge as way more bitcoin is being added to the ecosystem.

There are two ways I implemented proof-of-work adjusting over time. The first is simply increasing the difficulty every n number of blocks. In my implementation, I did it every five blocks. Implementing it like this is to directly see the proof-of-work getting harder in a system

```
// Target increasing every n blocks. Better for example where power doesn't change.
//
this.target = target;
if(this.chainLength != 0 && this.chainLength % 5 == 0) {
  console.log("DIFFICULTY INCREASING");
}
this.target = this.target >> BigInt(Math.floor(this.chainLength / 5));
//
```

where the number of miners doesn't increase. The second implementation is closer to Bitcoin as it increases or decreases difficulty based on the time between the last two blocks. I did this by storing the previous two blocks' timestamps and target every time a new block is created. Then based on the seconds between those timestamps, I increased the difficulty if it was faster than 10

seconds and decreased the difficulty if it was slower than 10 seconds. This SpartanGold implementation now has a self-adjusting proof-of-work feature like Bitcoin.

```
// This is for difficulty adjustment based on time.
//
// This is to get the last two targets trough the last blocks.
if(this.chainLength == 0) {
  // If its the first block initialize target, targets, and timestamps.
  this.target = target;
  this.timestamps = [];
}
else if(this.chainLength == 1 || this.chainLength == 2) {
  // If its block 1 just add to targets & timestamps, get previous target.
  this.target = prevBlock.target;
  this.timestamps = prevBlock.timestamps;
}
else {
  // If its any other block remove the oldest target/timestamp from targets/timestamps, get the previous target.
  this.target = prevBlock.target;
  this.timestamps = prevBlock.timestamps;
  this.timestamps.shift();
}
// push newest target to the list.
this.timestamps.push(this.timestamp);

// Seconds between the previous block and the block before it.
let secondsBetween = (((this.timestamps[1] - this.timestamps[0]) % 60000) / 1000);
// Make it easier if its over 10 seconds.
if(secondsBetween > 10) {
  this.target = this.target << 1n;
}
else if(secondsBetween < 10) {
  // Make it harder if its under 10 seconds.
  this.target = this.target >> 1n;
}
```

Conclusion

All in all, I implemented features from Bitcoin like Merkle tree, fixed block size, and proof-of-work adjustment over time to SpartanGold. This project allowed me to learn a lot more about what it takes to implement blockchain features and made me realize how different cryptocurrencies can be. During this project, the challenges I faced were mostly from trying to plug in the Merkle tree in place of the SpartanGold's Map with all the different functions Map uses. Furthermore, making the proof-of-work adjust overtime was also pretty tough as SpartanGold blocks' previous block value isn't always the proper last block in the chain. Lastly,

there weren't any actual miners with RAM to store Merkle tree paths. Nobody is validating transactions with this sample SpartanGold Cryptocurrency, so I resorted to keeping transactions in the Merkle tree class. In the future, I will keep exploring new features as blockchain technology keeps advancing and new cryptocurrencies keep coming out. Seeing if these features are good, helpful, or vulnerable is a blast.