Project 4: Bitcoin

This project is due on **April 1, 2019** at **11:59pm CST**. We strongly recommend that you get started early. You will get 80% of your total points if you turn in up to 72 hours after the due date. Late work will not be accepted after 72 hours past the due date.

The project is split into two parts. Checkpoint 1 helps you to get familiar with the language and tools you will be using for this project. The recommended deadline for checkpoint 1 is **March 15, 2019**. We strongly recommend that you finish the first checkpoint before the recommended deadline. However, you do NOT need to make a separate submission for checkpoint 1. That is, you need to submit your answers for both checkpoints in a single folder before the project due date on **April 1, 2019** at **11:59pm CST**. Detailed submission requirements are listed at the end of the document.

This is a group project; you SHOULD work in a group of **2 or 3 students**.

The code and other answers you submit must be entirely your own work, and you are bound by the Student Code. You MAY consult with other students about the conceptualization of the project and the meaning of the questions, but you MUST NOT look at any part of someone else's solution or collaborate with anyone else. You may consult published references, provided that you appropriately cite them (e.g., with program comments), as you would in an academic paper.

Solutions MUST be submitted electronically on GitLab, following the submission checklist given at the end of each checkpoint. Details on the filename and submission guideline is listed at the end of the document.

ONE submission per group is sufficient.

Release date: March 7, 2019
Checkpoint 1 Recommended Due date: March 15, 2019
Checkpoint 2 Due date: April 1, 2019 at 11:59pm CST

Course GitLab: https://gitlab.engr.illinois.edu/cs-463-spring-2019/YourNetId

Introduction

Bitcoin has become an increasingly popular cryptocurrency in recent years. In Bitcoin, all user transactions are kept in a public ledger, and users are only identified by cryptographic public keys. This provides some level of anonymity. In this machine problem, you will study the Bitcoin blockchain structure, and query blockchain data with online Bitcoin developer APIs. You will also study the level of anonymity in a Bitcoin system, and demonstrate that it is possible to link multiple Bitcoin addresses to the same user, and infer some users' identities.

Please read the assignment carefully before you start implementing.

Objectives

- Understand the Bitcoin blockchain structure.
- Be able to analyze Bitcoin blockchain data and get reasonable information.
- Gain familiarity with Bitcoin APIs in Java.

Checkpoints

This machine problem is split into 2 checkpoints. Checkpoint 1 will help you get familiar with the Bitcoin developer APIs provided at Blockchain.info. You will need to answer a few questions by querying a block of transactions with the provided APIs. In Checkpoint 2, you will implement an algorithm to cluster Bitcoin addresses that are likely to belong to the same user, generate a user graph of transactions within a single day, and answer some questions based on the user graph.

Implementations

To help you with the implementation, you are provided with a code skeleton which prototypes the main classes and includes some useful methods. You are free to modify the provided code as long as the main classes in the `test` package are preserved.

4.1 Checkpoint 1 (20 points)

This machine problem is split into 2 checkpoints. Checkpoint 1 will help you get familiar with the Bitcoin API. You will need to complete the implementation for `Checkpoint1.java`, query the Bitcoin block chain with the provided APIs, and answer a few questions. `Checkpoint1Test.java` will write your answers into a file named `cp1.txt`. Please DO NOT modify `Checkpoint1Test.java`.

4.1.1 Block Info API

BlockInfo (`https://blockchain.info/`) is a Bitcoin block explorer website that provides information about Bitcoin blocks, transactions, and addresses. It also provides a Java API for querying this information. You can find the source code and documentation for these APIs on GitHub (`https://github.com/blockchain/api-v1-client-java`). The Java library (`api-1.1.0.jar`) is included in the provided code skeleton.

4.1.2 Blocks (10 points)

In Bitcoin, transactions are stored in files called blocks. Each block contains a list of transactions and a block header.
Get the block with hash "000000000000000f5795bfe1de0381a44d4d5ea2ad81c21d77f275bffa03e8b3", and answer the following questions:

1. What is the size of this block?
2. What is the Hash of the previous block?
3. How many transactions are included in this block?

Tips:

- Complete the implementation of the constructor in `Checkpoint1.java`. Use the method `getBlock(String hash)` in `BlockExplorere.java` to get a `Block` object in the block chain with the corresponding hash.
- Complete the implementation of `getBlockSize()` in `Checkpoint1.java`. Use the method `getSize()` in `Block.java`.
- Complete the implementation of `getPrevHash()` in `Checkpoint1.java`. Use the method `getPreviousBlockHash()` in `Block.java`.
- Complete the implementation of `getTxCount()` in `Checkpoint1.java`. Use the method `getTransactions()` in `Block.java`.

What to submit Submit together with 4.1.3.

4.1.3 Transactions (10 points)

In Bitcoin, each transaction has a list of inputs and a list of outputs. In a normal transaction, an input is a reference to an output from a previous transaction. It contains a hash of the previous transaction and an index indicating the specific output of that transaction. An output contains a value and a Bitcoin address. The value shows the number of Satoshi (1 BTC = 100,000,000 Satoshi) to be transferred, and the Bitcoin address shows who should receive the transferred Bitcoins. A

generation transaction (i.e. coinbase transaction) refers to a transaction that generates new Bitcoins. A generation transaction has a single input. The input has a "coinbase" parameter, and has no previous outputs.
Get all the transactions in the Bitcoin block in 4.1.2, and answer the following questions:

1. Find the transaction with the most outputs (if there are several transactions with the same number of outputs, choose the first transaction), and list the Bitcoin addresses of all the outputs.
2. Find the transaction with the most inputs (if there are several transactions with the same number of inputs, choose the first transaction), and list the Bitcoin addresses of the previous outputs linked with these inputs.
3. Which Bitcoin address has received the largest amount of Satoshi in a single transaction?
4. How many coinbase transactions are there in this block?
5. What is the number of Satoshi generated in this block?

Tips:

- In `Transaction.java`, the method `getInputs()` returns a list of `Input` objects; the method `getOutputs` returns a list of `Output` objects.
- Complete the implementation of `getOutputAddresses()` in `Checkpoint1.java`. To get the Bitcoin address of an Output object, use method `getAddress()` in `Output.java`.
- Complete the implementation of `getInputAddresses()` in `Checkpoint1.java`. To get the previous output of an Input object, use method `getPreviousOutput()` in `Input.java`.
- Complete the implementation of `getLargestRcv()` in `Checkpoint1.java`. Hint: To get the number of Satoshi received by an Output object, use method `getValue()` in `Output.java`.
- Complete the implementation of `getCoinbaseCount()` in `Checkpoint1.java`. In a coinbase transaction, there is a single input, and the input has no previous output (i.e., `getPreviousOutput() == null`).
- Complete the implementation of `getSatoshiGen()` in `Checkpoint1.java`.
- Compile and run the code with `compile.sh` and `run1.sh`. This will write the answers for all the questions in Checkpoint1 to a file named `cp1.txt`.

What to submit

- `cp1.txt` The file generated by `Checkpoint1Test.java`.
- Implementations for this section

4.2 Checkpoint 2 (80 points)

For this checkpoint, you'll need to query all the transactions on 10/25/2013, generate a user graph based on the transactions, and analyze the user graph.

4.2.1 Generate a Transaction Dataset (10 points)

Implement `DatasetGenerator.java` to generate a dataset for all the **non-coinbase** transactions on 10/25/2013. The dataset should have one record for each input or output in a transaction. Each record should have 5 fields:

- `txIndex` The index of the transaction *Hint: Use method* `getIndex()` *in* `Transaction.java`
- `txHash` The hash of the transaction *Hint: Use method* `getHash()` *in* `Transaction.java`
- `address`
  - For an input record, this is the Bitcoin address of the previous output
  - For an output record, this is the Bitcoin address of the output
- `value`
  - For an input record, this is the value of the previous output
  - For an output record, this is the value of the output
- `in/out` Indicate whether this is an input record or an output record

Use the method `generateInputRecord` and `generateOutputRecord` to generate each record in the dataset. After finishing your implementations, use `compile.sh` and `run_gen.sh` to write the dataset to file `transactions.txt`. An example of the generated dataset is provided in `transactions_eg.txt`. The example dataset contains all the non-coinbase transactions in the block we studied in 4.1.2.

Tips

- The process of downloading the transactions may take a few seconds to a few minutes depending on the network bandwidth.
- Please use `blockExplorer.getBlocksAtHeight()` to get blocks with height between **[265852, 266085]**. This contains all the blocks generated on 10/25/2013 UTC. Please do **NOT** use `blockExplorer.getBlocks(long timestamp)`. Although this method does return the blocks generated on a certain date, it returns at most 200 blocks, which is not a complete list of all the blocks.

What to submit

- `transactions.txt` A dataset containing all the non-coinbase transactions on 10/25/2013, generated by `DatasetGeneratorTest.java`.
- Implementations for this section

4.2.2 Cluster Bitcoin Addresses (20 points)

Joint control is a common idea used to cluster addresses. It assumes that addresses used as inputs to a common transaction are controlled by the same entity. Implement `UserCluster.java` to cluster Bitcoin addresses based on joint control, and assign a unique userId for each cluster of addresses. The `UserCluster` should have at least 2 attributes. `Map<Long, List<String» userMap` maps a user id to a list of Bitcoin addresses, and `Map<String, Long> keyMap` maps a Bitcoin address to a user id. After finishing your implementations, use `compile.sh` and `run_cluster.sh` to write `userMap` and `keyMap` to a file, generate the answers for the following questions, and generate a user graph file:

1. How many users (number of clusters) do you get after clustering?
2. What is the size of the largest cluster?

Tips:

- To start, implement the method `readTransactions()` to read all the transactions in the dataset.
- Implement the method `mergeAddresses()` to merge addresses that are used as inputs to a common transaction, and store them in `userMap` and `keyMap`
- Both `userMap` and `keyMap` should contain ALL Bitcoin addresses that appeared in the inputs and outputs of non-coinbase transactions.
- Implement the method `writeUserGraph()` to generate a user graph file based on transactions in the dataset. The file should contain the input userId, output userId, and value of transferred Bitcoin (in Satoshi) for each output. `userGraph_eg.txt` is an example of the user graph file. The structure of your user graph file can be different from the example. You can also include other additional information if you need. You will use this user graph file to do graph analysis in 4.2.3
- You can use `transactions-test.txt` to test your clustering algorithm. Feel free to design other test cases by changing the transactions in `transactions-test.txt`.

What to submit

- `cp2.txt` Answers for the questions.
- `userMap.txt` A map from userId to Bitcoin addresses.
- `keyMap.txt` A map from Bitcoin addresses to userId.
- `userGraph.txt` The user transaction graph
- Implementations for this section

4.2.3 Analyze User Graph (10 points)

On 10/25/2013, the FBI seized $28.5 million in Bitcoins from Ross Ulbricht, the alleged owner of silk road. This seizure was done with multiple transactions, which are recorded in the blockchain. Please find the Bitcoin address(es) that are controlled by the FBI by analyzing the user graph you generated in 4.2.2. Then, find at least 3 Bitcoin addresses that you think might belong to an owner or a user of the Silk Road based on your analysis. You can assume that it is unlikely for other users to receive a similar or larger amount of transactions in a single day. You can use search results from online Bitcoin explorers (e.g. `https://blockchain.info/`) to check whether your guess is correct.
To analyze the user graph, you can use open source graph APIs or your own implementations. Please include all of your implementations and a `ReadMe.txt` file in your submission. The `ReadMe.txt` file should give detailed instructions on how to run your code.

What to submit

- `ReadMe.txt`
- Implementations for this section

4.2.4 Report (50 points)

Please write a 1-page report as follows:

- Put the names and netids of ALL the group members at the top of the report. (5 points)
- Briefly describe the clustering algorithm you used in 4.2.2. (10 points)
- Is the assumption of *joint control* a valid assumption? Why or why not? Can you think of any false positives and false negatives for the clustering algorithm based on this assumption? (10 points)
- What is the Bitcoin address controlled by the FBI in 4.2.3? Briefly describe the algorithm you used to identify this address. (15 points)

- List at least 3 Bitcoin addresses that you think might belong to an owner or a user of the Silk Road based on your analysis. Why do you think so? (This is an open-ended question. There is no "correct" answer for it. Be creative and try to propose strong arguments for your guess.) (10 points)

Please name the report as `report_[netid1]_[nedid2]_[netid3].pdf`.

What to submit

- `report_[netid1]_[nedid2]_[netid3].pdf`

Submission Checklist

Create a project called mp4 in your GitLab repository.
Place the following files in your mp4 GitLab project.

- `report_[netid1]_[nedid2]_[netid3].pdf` [Report for Section 4.2]
- `ReadMe.txt` [4.2.3]
- `cp2.txt` [Answers for the questions in 4.2.2]
- `userMap.txt` [A map from userId to Bitcoin addresses in 4.2.2]
- `keyMap.txt` [A map from Bitcoin addresses to userId in 4.2.2]
- `userGraph.txt` [The user transaction graph in 4.2.2]
- `transactions.txt` [A dataset containing all the non-coinbase transactions on 10/25/2013, generated by `DatasetGeneratorTest.java`]
- `cp1.txt` [The file generated by `Checkpoint1Test.java`]
- `src` [Implementations for both checkpoints]
- `lib` [Libraries]
- `bin` [Compiled java classes]
- `compile.sh` [Provided compilation script]
- `run1.sh` [Provided script for checkpoint1]
- `run_gen.sh` [Provided script for 4.2.1]
- `run_cluster.sh` [Provided script for 4.2.2]