Project 3: Bitcoin

This project is due on **October 22nd, 2024** at **11:59pm CST**. We strongly recommend that you get started early. We strongly recommend that you get started early. You have a total of 3 grace days for the 5 MPs, which allow for a 24-hour extension on your MP deadlines. You can use up to 2 grace days per MP, but no more than 3 in total across all MPs. If you submit your assignment after the due date and beyond the grace day period, you will lose 20% of the total points for each late day.

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 **October 15th, 2024**. We strongly recommend that you finish the first checkpoint before the recommended deadline. You need to submit your answers for both checkpoints in a single folder before the project due date on **October 22nd, 2024** at **11:59pm CST**. Detailed submission requirements are listed at the end of the document.

This is an individual project; you SHOULD work **individually**.

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.

This semester we have anti-cheating check on our autograder, so please do not take risk to cheat. We will routinely check for plagiarism. WE NEVER TOLERATE ANY CHEATING, no matter for cheating or being cheated.. You will receive one warning if any is detected, and repeat offenses will result in a zero on the corresponding MP and a report to the appropriate authorities.

Solutions MUST be submitted electronically on PrairieLearn and Github, 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.

Release date: October 08th, 2024
Checkpoint 1 Recommended Due date: October 15th, 2024
Checkpoint 2 Due date: October 22nd, 2024 at 11:59pm CST

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, in order to do data analysis based on the user graph.

Implementations

To help you with the implementation, you are provided with a code skeleton that prototypes the main classes and includes some useful methods. You are free to modify the provided code in **./src/main** as long as the main classes in the `test` package and those **.sh** files are preserved. Please don't use other java packages. The provided packages in the code skeleton should be sufficient for this MP. **Please make sure .sh files can be executed properly to generate output.**

3.1 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`.**

3.1.1 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.

3.1.2 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 `BlockExplorer.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 3.1.3.

3.1.3 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 3.1.2, 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

- `Checkpoint1.java`

3.2 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 then you can use it to analyze the user graph.

3.2.1 Generate a Transaction Dataset (40 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 4 fields:

- `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 3.1.2.

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`. Please generate this txt file locally and upload it in your repo.
  - You only need to submit this on Github
- `DatasetGenerator.java`
  - You should submit this on both PrairieLearn and Github.

3.2.2 Cluster Bitcoin Addresses (40 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 can use this user graph file to do graph analysis.
- 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

- `UserCluster.java`
  - You should submit this on both PrairieLearn and Github.

3.3 Autograder

We have set up an autograder on PrairieLearn. To use it, you must first enroll in the course CS 463: Computer Security II by visiting `https://us.prairielearn.com/pl`.

After enrolling, navigate to the **Assessments** section within CS 463: Computer Security II, where you will find **MP3** here:
`https://us.prairielearn.com/pl/course_instance/163006/assessment/2455811`

Upload your `Checkpoint1.java`, `DatasetGenerator.java`, and `UserCluster.java` to the corresponding checkpoints to receive your scores.

Important Note

1. Your highest grade and grading report will be automatically recorded in our server.
2. You **must** submit through PrairieLearn to our auto-grader **at least once** before the deadline to get a grade. We don't grade your code manually, the only way is to use our auto-grader.
3. Try auto-grader as early as possible to avoid the heavy traffic before the deadline. **DON'T DO EVERYTHING IN THE LAST MINUTE!**
4. If the auto-grader is down for a while or you are encountering weird auto-grader behaviour, don't be panic. Contact us on Campuswire, we will fix it and give some extension if necessary.
5. We have anti-cheating mechanisms in place. **NEVER CHEAT OR HELP OTHERS CHEAT!**
6. Please use Java 8 to test locally. Don't use any Java modules except for the ones that are specified in the release.
7. Never abuse the auto-grader. You should only submit your own code when using the auto-grader on PrairieLearn. Please do not abuse the auto-grader in any form. Abusing the auto-grader is considered a form of violation of the student code of conduct, and the corresponding evidence will be gathered and reported.
8. You only have 10 min for each checkpoint. Please be aware of this and improve the algorithm efficiency.
9. When uploading your code to PrairieLearn, please remove any debug messages printed to the console. Otherwise, it may cause your submission to fail the test cases in our autograder.

3.4 Submission Instructions

There are two tasks you need to complete for submitting your work:

1. Upload to PrairieLearn

You must upload your code to PrairieLearn, which serves as our autograder. Specifically, upload `Checkpoint1.java`, `DatasetGenerator.java`, and `UserCluster.java` to **MP3 Checkpoint 1** and **MP3 Checkpoint 2** on PrairieLearn. It is essential to complete these uploads to receive and record your MP3 score.

Submission Limit

Please note that you are limited to a maximum of 30 submissions for each checkpoint on PrairieLearn. Be mindful of each submission you make, as your submission count is limited.

Timeout

The MP3 autograder on PrairieLearn has a timeout limit of 600 seconds. If your code exceeds this time limit, the autograder will throw an error and assign a score of 0 for that submission. Ensure your algorithm is time-efficient to avoid timeouts.

2. Upload to GitHub

In addition to uploading to PrairieLearn, you must also upload your code to GitHub. After the assignment deadline, the course staff will compare the code you submitted to PrairieLearn with the code you uploaded to GitHub to ensure they match. We will also perform anti-cheating checks to detect any plagiarism. Please note that any form of code plagiarism is strictly prohibited, including copying parts of someone else's code or code from previous students.

Folder Structure

- Make sure there is a folder "mp3" inside the root folder of your Github repository in the main branch. Your score will be deducted if you fail to do so.
- Don't modify libs and test (three test files). They are provided in code skeleton.
- Generate **transactions.txt** locally and directly put the generated text file into your mp3 folder.
- You still need to upload your **DatasetGenerator.java** file for us to check you are not copying someone's **transactions.txt**.
- Upload `interaction_log.txt` if you used AI chat tools to assist you with the MP. Include the interaction logs of your conversations with AI tools in this file. If you did not have conversations with AI tools for the MP, you can ignore this.

In conclusion, your folder structure for MP3 should be as follows:

```
mp3
├── libs
│   ├── api-1.2.0.jar
│   ├── commons-lang3-3.12.0.jar
│   └── gson-2.2.jar
├── src
│   ├── main
│   │   ├── Checkpoint1.java
│   │   ├── DatasetGenerator.java
│   │   └── UserCluster.java
│   └── test
│       ├── Checkpoint1Test.java
│       ├── DatasetGeneratorTest.java
│       └── UserClusterTest.java
├── transactions.txt
└── interaction_log.txt (if you have conversations with AI tools for MP3)
```

Git Upload

Use the following commands to push your submission for grading (it is advisable to do this frequently for better version control). Grading will be based on the latest submission to the autograder before the deadline. Please push to your main branch rather than creating a separate branch.

```
git add -A
git commit -m "REPLACE THIS WITH YOUR COMMIT MESSAGE"
git push origin main
```