Project 3: Bitcoin

This project is due on **March 20th, 2024** 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. Please let the TAs know if you decide to take this option (email or private campuswire post). 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 13th, 2024**. 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 **March 20th, 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 don't take risk to cheat. WE NEVER TOLERATE ANY CHEATING, no matter for cheating or being cheated.

Solutions MUST be submitted electronically on Github repository, 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: February 26th, 2024
Checkpoint 1 Recommended Due date: March 13th, 2024
Checkpoint 2 Due date: March 20th, 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, and answer some questions 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 analyze the user graph.

3.2.1 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 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.
- `DatasetGenerator.java`

3.2.2 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 3.2.3
- 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

- `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.
- `UserCluster.java`

3.2.3 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 3.2.2. 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

3.2.4 Answer Questions (40 points)

- For this part, please go to Coursera Week 7 "MP3: Bitcoin Analysis" to answer all the questions. **Please note that you can make an one-time submission only**.

What to submit

- Please go to Coursera to submit your answers.

3.3 Submission

3.3.1 Folder Structure:

- Please ensure your root folder named "mp3"
- Don't need to 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. For 3.2.1 we will grade directly based on your uploaded transactions.txt.
- You still need to upload your DatasetGenerator.java file for us to check you are not copying someone's transactions.txt.
- For 3.2.3, put a ReadMe.txt and all implementations into a folder named "Implementation_for_3_2_3". Write down everything we need to know in ReadMe.txt so that we can reproduce your result. No strict folder structure requirement in this section, just ensure everything is under folder "Implementation_for_3_2_3".
- Auto-grader will not grade your 3.2.3. We will manually grade it after the deadline.

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
├── Implementation_for_3_2_3
│   ├── ReadMe.txt
│   └── YourImplementations
└── transactions.txt
```

3.3.2 Git Upload

Then use the following commands to push your submission for grading (do this often for better version control). The grading will be based on the latest auto-grader submission before the deadline. Please push into your main branch rather than a self-created one.

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

3.3.3 Auto-grader

This semester we use auto-grader to eliminate any misalignment between student's working and grading.

Please use Java8 to test locally.

To use the auto-grader please do as follows:

1: Upload your files into your git repo as mentioned above. Please follow section 3.3.1 to have a correct folder structure.

2: Send HTTP POST request with your netid. This one is similar to previous 2 MPs.

We provide 2 methods, pick one as your preference. We recommend command line because it's more reliable.

Method1: Command Line

To grade cp1:

```
curl -X POST -H "Content-Type:  application/json" -d @submission.json
http:/128.174.136.25:8080/grade_cp1
```

To grade cp2:

```
curl -X POST -H "Content-Type:  application/json" -d @submission.json
http:/128.174.136.25:8080/grade_cp2
```

`submission.json` is a json file in your local computer, with only one key-value pair "netid":"YOUR_NETID"

Replace "YOUR_NETID" with your own netid, such as "netid":"aj3".
It's fine to change "submission.json" into other names, just remember to ensure it is a json file and you are doing the same change to the command line.
Remember, it's HTTP request rather than HTTPS.

Method2: Web Tool

If you don't want to use the above command line (sometimes more nasty), you can also use web tool such as Postman to send HTTP POST request to http:/128.174.136.25:8080/grade_cp1 or http:/128.174.136.25:8080/grade_cp2 with a json body carring a key-value pair, "netid":"YOUR_NETID"

Replace "YOUR_NETID" with your own netid, such as "netid":"aj3".
Remember, it's an HTTP request rather than HTTPS.

3: Get Feedback

The feedback will be a web response to your HTTP request. The format is as follows:

```
{
  "grade report":
  "=============3.1===========
    3.1.2
    1.correct
    2.correct
    3.correct
    3.1.3
    1.correct
    2.correct
    3.correct
    4.correct
    5.correct

score: 20/20
",
  "run feedback":
  "------------------------------
    start 3.1 grader compilation
    Compilation succeded!
    ------------------------------
    checking zz45
    ------------------------------
    start student compilation: run compile.sh
    compile.sh finished
    ------------------------------
    start student run: run run1.sh
    run1.sh finished
    ----------------------
    start 3.1 testing
    Run succeded!
  "
}
```

or for cp2:

```
{
```

```
  "grade report":
  "=============3.2.1==========
    match ratio: 0.9999930660287413
    score: 10/10
   =============3.2.2==========
    3.2.2 1: correct
    3.2.2 2: correct

   score: 20/20",
  "run feedback":
  "------------------------------
    start 3.2.2 grader compilation
    Compilation succeded!
    ------------------------------
    checking zz45
    ------------------------------
    start student compilation: run compile.sh
    compile.sh finished
    ------------------------------
    start 3.2.1 grader compilation
    start 3.2.1 testing
    Run succeded!
    ------------------------------
    start student run: run run_cluster.sh
    run_cluster.sh finished
    ---------------------------
    start 3.2.2 testing
    Run succeded!
    "
}
```

Any error will be put into "run feedback" field such as Java error trace, wrong folder structure, git error or wrong web request. Runtime info will also be put into this field.
Your grading result and grade for this checkpoint will be put into the "grade report" field. Remember, your grade and grading report will be automatically recorded on our server.

If you see **"curl: (6) Could not resolve host: application"**, don't worry, just wait for the result.

3.3.4 Timeout

You have 2 min for checkpoint1 and 10 min for checkpoint2 (in total 12 min). Auto-grader will throw an error and 0 point if timeout value is reached. Again, we won't grade your 3.2.3 through auto-grader.

The time spent creating the environment and downloading git is not counted in timeout.

3.3.5 Abuse

You should only submit your own NetID when querying the auto-grader. Please do not abuse the auto-grader in any form. Abuse behaviors include but are not limited to using other students' NetIDs to submit a query and get their grading results. We will closely monitor the NetIDs sent by each IP address. 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.

3.3.6 Important Note

1: Your highest grade and grading report will be automatically recorded in our server.

2: You **must** send request 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!**. This time we have increased capacity of our auto-grader, because we notice many students complaining the auto-grader is slow when close to deadline. Now it will be much faster than MP1 and MP2, but still please do it ASAP.

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: If you see **"curl: (6) Could not resolve host: application"**, don't worry, just wait for the result to show up.

6: We have anti-cheating mechanisms, NEVER CHEAT OR HELP OTHERS CHEAT!

7: If you are using web tool such as Postman, sometimes you will fail and see "server hang up". Keep trying more times or use command line instead.

8: Please use Java8.

9: Read section 3.3.5 carefully. Never abuse the auto-grader.

10: You only have 10 min for CP2 (3.2.3 not included). Please be aware of this and improve the algorithm efficiency.