Max Goldstein

CS 5450

Report for Part 1

For this project, I decided to use Java as my programming language for a few reasons, the main reason being that I am working for JPMorgan after I graduate Java is very widely used there so I wanted a refresher. I did not use any IDE like Netbeans, I just used my favorite editor Visual Studio Code and made, managed, and compiled files manually using the command line. The Java version running on my machine is 9.0.4.

The distributions I implemented were uniform, skewed, and four-tiered(I treated skewed as my choice distribution as discussed in class). For the uniform distribution, I implemented a generic Fisher-Yates shuffle, which creates a pseudo-random permutation of the finite list of classes. For the skewed distribution, I tweaked the generic Fisher-Yates algorithm so that instead of picking a random number between “i” and the length of the class list, it picks the minimum of two random numbers between “i” and the length of the class list, hence linearly skewing the course distribution. Lastly, for the four-tiered distribution, I again modified the generic Fisher-Yates algorithm so that it picks a random number between 0-99. If that number if less than 40, then it does Fisher-Yates on the first quarter of the class list. If it is between 70 and 40 then it uses Fisher-Yates to pick from the second quarter of the list, and so on. Below are histograms of the courses created. Note: because Jupyter Notebooks (Python) is very easy to create graphs and histograms, I adapted my Java code to Python and created these graphs in Jupyter Notebooks

A screenshot of a cell phone

Description automatically generatedA screenshot of a social media post

Description automatically generatedA screenshot of a cell phone

Description automatically generated

For time and space complexity, I will go through the following code snippets to prove my hypothesis:

A picture containing drawing

Description automatically generated

A close up of a logo

Description automatically generated

This constructor initializes and sets the array of all of the classes.

Current space complexity: O(c), where c is the total number of classes

Current time complexity: O(c)

A picture containing drawing

Description automatically generated

A screenshot of a cell phone

Description automatically generated

A new array of size k is initialized to store the assigned classes, bringing the space complexity to O(c + k). Next, the function loops k number of times doing a constant amount of work and not using any more memory. However, this function just gets the assigned classes for one student, so it is called s times. Making the final complexities for the uniform distribution as follows:

Space complexity: O(c + k)

Time complexity: O(c + s\*k)

A picture containing black, drawing

Description automatically generated

A screenshot of a cell phone

Description automatically generated

The important difference to note between this and the uniform distribution is that the array of all classes needs to be reformatted after each call. With the uniform distribution, it did not matter if the allClasses array was out of order because every number still had an equally likely chance of being chosen so it didn’t matter if the class number didn’t match the index of the class array. Now, each class needs to be in the correct index so that when randVal is chosen, it is actually getting that class, therefore linearly skewing the data. Because of this reformatting that happens, the time complexity is increased by a factor of c for each time it is called. However, the space complexity stays the same, so the complexities are:

Space: O(c + k)

Time: O(c\*s\*k)

A picture containing drawing

Description automatically generated



The complexities for the 4-tiered distribution are the same as the skewed distribution because it require only a constant amount of more space and time.

Space: O(c + k)

Time: O(c\*s\*k)

Upon further analysis, my four-tiered distribution seems to have a bug causing it not to run properly (when I thought it was running four-tiered it was really running uniform, the biggest mystery is that my python adaptation runs correctly). Given that this project is already late, I have decided to leave it out of the remainder of the report and only showcase the uniform and skewed distributions.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| C | S | K | Time(ms) Uniform | Time(ms) Skewed |
| 1,000 | 10,000 | 10 | 7 | 14 |
| 1,000 | 100,000 | 10 | 26 | 67 |
| 10,000 | 100,000 | 10 | 29 | 349 |
| 10,000 | 100,000 | 20 | 37 | 365 |

I took a slightly different approach to counting conflicts as I was having trouble doing it in O(m) space. I begin with a string array eTemp, which is of length c and each cell is an empty string, this takes O(c) time and O(c) space to prepare. Each index will hold the conflicts for that class as a string of numbers separated by dashes i.e “23-42-12-54”. Next, I generate the classes for a student, which I will ignore the space and time complexities of since those were discussed above and it could be done elsewhere but for my implementation it made the most sense to generate the courses and count conflicts at the same time. Next, the function enters a loop that iterates through all of the classes generated for this student. For each class, it retrieves the string in eTemp that hold the conflicts for that class and splits the string by the dashes, so I am left with an array of the conflicts for that course. Unfortunately, I scoured Java documentation but was not able to find anything in regards to complexities or actual implementation details for the split function. Now that I have a class and its previous conflicts, I enter a second loop that iterates through the other classes for this student, and at each element I iterate through the list of previously seen conflicts and if it already exists then I move on to the next class, and if it doesn’t exist in the previously seen conflicts, then I add it to the string of conflicts for the current class. I will summarize with pseudocode because it can be a bit confusing.

For each student

For each class, c1, taken by student

For each other class, c2, taken by student

For each previously seen conflict of c1

If c2 is already a conflict of c1, do nothing, else add c2 to conflicts of c1

So the time complexity of this is on the order of O(c + s\*k\*k\*c) = O(c + s\*c\*k^2), however the space complexity is O(m) because no duplicate conflicts are ever added. This runtime is very slow for this operation, but it was the only way I could think of to get it to run in O(m) space. Had there been a time constraint instead of a memory constraint, I would have used a 2d boolean array that was c by c with all values initialized to false. To add a conflict while making sure it was not a duplicate, I would have done the following: if the two conflicting courses were X and Y, then I would have gone to the index [X][Y] and if it was false then then I would have made it and [Y][X] true. If it was already true, then it is a duplicate conflict and nothing needs to be done. This would have been much faster than the implementation I went with, however it requires O(c^2) space which is greater than the O(m) space requirement.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| c | s | k | distribution | time(ms) |
| 200 | 1000 | 20 | uniform | 883 |
| 200 | 2000 | 20 | uniform | 1512 |
| 200 | 2000 | 40 | uniform | 5095 |
| 200 | 4000 | 20 | uniform | 2846 |
| 400 | 2000 | 20 | uniform | 2478 |

Some simple test cases to show that part 1 is functional: A screenshot of a cell phone

Description automatically generated A screenshot of a cell phone

Description automatically generated

Part 2