**Project 2 Report**

**Group #3**

Language: Java

Da-El Kim

Hang Phan

Alibek Zeinolla

Olufisayo Ayodele

Alikhan Adilbekov

**Outline**

Interpretation of the Given Problem…………………………………….3

Methodology of the Solution……………………………………………4

Experimental Results……………………………………………………6

Conclusion………………………………………………………………8

**Interpretation of the Given Problem**

In the real life, we have many things to do. We always need think of the sequence to get things done: which we need do first, which we can do next.

In this project, we were given major courses that were needed for Computer Science majors and their prerequisites. We are to write code to solve the topological ordering problem of what classes to take next in order to finish our program quick. We need think about the sequence of taking the courses, since some courses have their prerequisites.

Basically, we are to design, implement, and test an abstract data type (ADT) graph in its prototype. Our project should include frequently used algorithms such as DFS, BFS, and so on. We implement topological ordering algorithm that’s based on DFS in order to output the sequence of courses. Then a sorting algorithm that puts the courses in the order according to their post visiting numbers.

**Methodology of the Solution**

A class named CourseGraph was created to handle the Depth First Search and Explore procedures. The main details about all the courses needed in the graph were stored in in a text file to be later used in the CourseGraph class. Data about each course and their prerequisites were extracted from the text file and stored in a Map, where the key of the Map would be a String (course symbol) and the value of the map would be an ArrayList of Strings (prerequisites of the course).

After the data is stored, the DFS algorithm is used to traverse the graph by decomposing the graph into its connected components. In the DFS, we call the explore method to explore the graph and find immediate connected components of the passed in key value.

Methods used in the project are further explained below:

**public** CourseGraph(): Constructor. It initializes all the private variables, and the Maps are initialized as TreeMaps.

**public** ArrayList<String> getAllKey(): Getter for all keys (course symbols) from the TreeMap in form of an ArrayList.

Input: no input.

Process: a for loop to store all the keys of the TreeMap in an ArrayList

Output: all keys in form of an ArrayList.

**public** ArrayList<String> getvaluelist(String key): Getter for the values (prerequisites) from the TreeMap in form of an ArrayList

Input: key (course symbol).

Process: a for loop to store all the prerequisites of the courses in an ArrayList

Output: all values in form of an ArrayList.

**public** **void** initialize(String myFileName): This method reads from the text file

Input: file name.

Process: read from the file and send each individual line to the splitLine method to be organized.

Output: no output.

**public** **void** splitLine(String line): This method organizes each line sent from the text file. It extracts only the data that we need from each line in the text file.

Input: a line from the text file.

Process: split the given line and extract important data like course symbol and prerequisites.

Output: no output.

**public** **void** setOriGraph(): This method Adds courses prerequisites to an ArrayList then put it in the map.

Input: no input.

Process: a nested for loop to add courses prerequisites to an ArrayList then put it in the map.

Output: no output

**public** **void** dfs(): Depth First Search method

Input: no input.

Process: set all visited nodes to false and store all key set in an array. Assign false for all visited elements. Then go through every key in the graph and explore the values of the key.

Output: no output

**public** **void** explore(String key, ArrayList<String> values): Explore method

Input: key (course symbol) that needs exploring and its values.

Process: get the index of the key and set its visit to true, increment its previsit. Loop through and explore unvisited nodes. Increment postvisit.

Output: no output

**public** **void** previsit(): is a method that just displays the total previsit.

**public** **void** postvisit(): is a method that just displays the total postvisit.

**public** **void** selectionSort(): is a method that sorts previsit, postvisit, and keys from highest to lowest to show the proper order that courses can be taken

Input: no input.

Process: set max to location 0, search the maximum element in the array, swap with the value at the max location, increment the max to point to the next element, repeat until the array is sorted

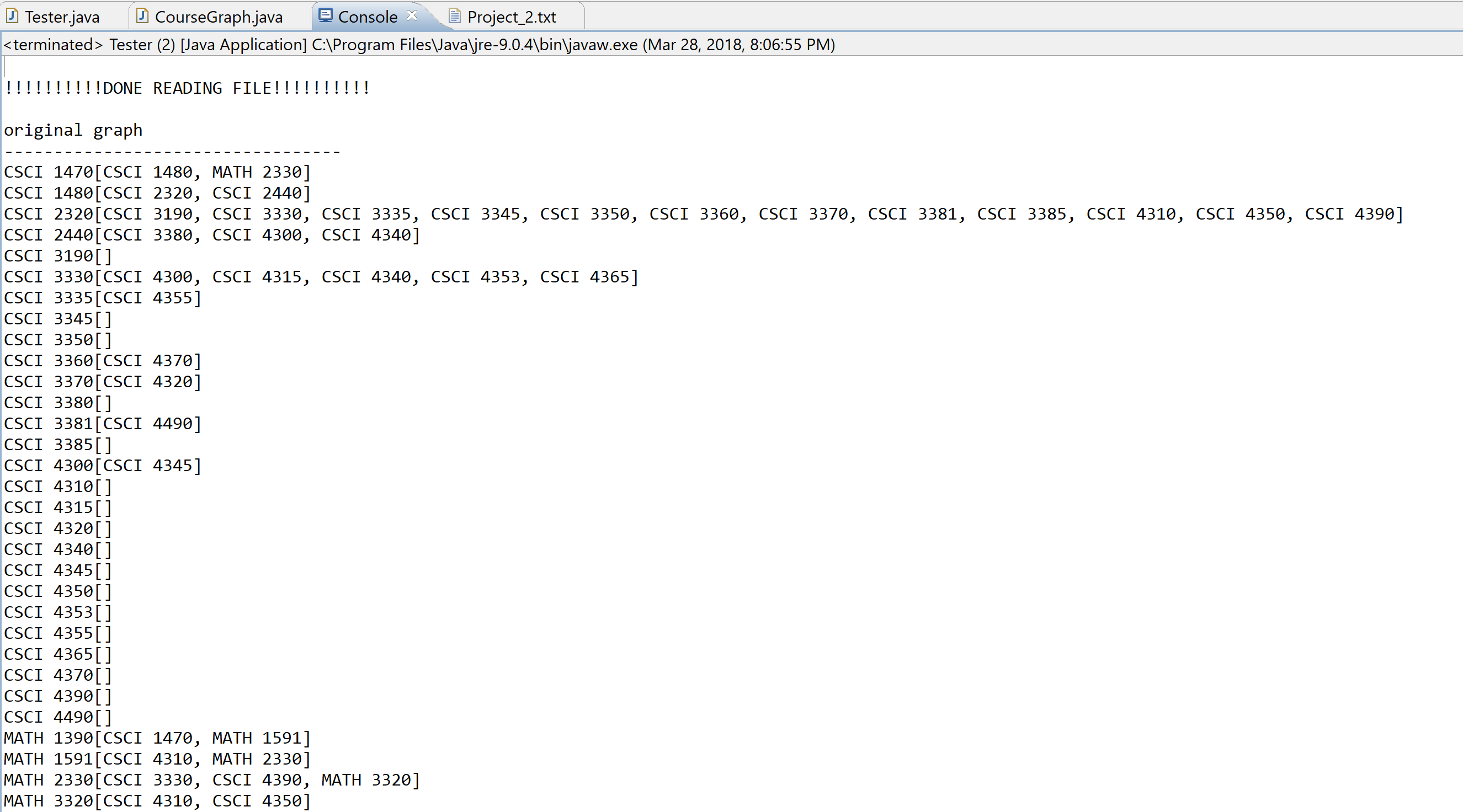
Output: no output

**public** **void** printOri(): is a method that prints the original graph.

**public** **void** print(): prints previsit and post visited order then sorts them by calling the selectionSort method.

**Experimental Results**

After the CourseGraph class was written, another class named Tester was created to test the CourseGraph class. The result of the test is below:

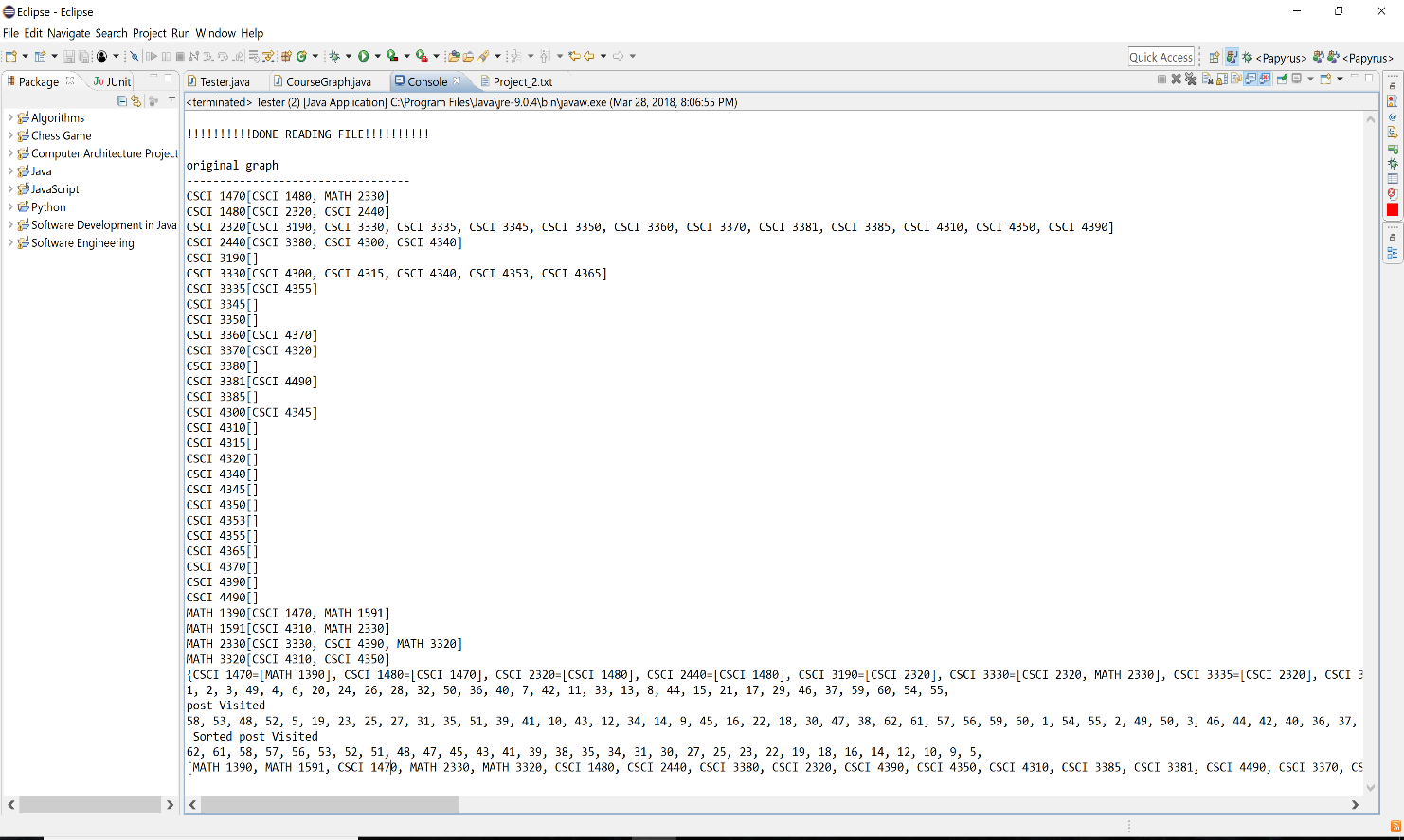
*Reverse Graph:* {CSCI 1470=[MATH 1390], CSCI 1480=[CSCI 1470], CSCI 2320=[CSCI 1480], CSCI 2440=[CSCI 1480], CSCI 3190=[CSCI 2320], CSCI 3330=[CSCI 2320, MATH 2330], CSCI 3335=[CSCI 2320], CSCI 3345=[CSCI 2320], CSCI 3350=[CSCI 2320], CSCI 3360=[CSCI 2320], CSCI 3370=[CSCI 2320], CSCI 3380=[CSCI 2440], CSCI 3381=[CSCI 2320], CSCI 3385=[CSCI 2320], CSCI 4300=[CSCI 2440, CSCI 3330], CSCI 4310=[CSCI 2320, MATH 1591, MATH 3320], CSCI 4315=[CSCI 3330], CSCI 4320=[CSCI 3370], CSCI 4340=[CSCI 2440, CSCI 3330], CSCI 4345=[CSCI 4300], CSCI 4350=[CSCI 2320, MATH 3320], CSCI 4353=[CSCI 3330], CSCI 4355=[CSCI 3335], CSCI 4365=[CSCI 3330], CSCI 4370=[CSCI 3360], CSCI 4390=[CSCI 2320, MATH 2330], CSCI 4490=[CSCI 3381], MATH 1390=[], MATH 1591=[MATH 1390], MATH 2330=[MATH 1591, CSCI 1470], MATH 3320=[MATH 2330]}

*PreVisit*: 1, 2, 3, 49, 4, 6, 20, 24, 26, 28, 32, 50, 36, 40, 7, 42, 11, 33, 13, 8, 44, 15, 21, 17, 29, 46, 37, 59, 60, 54, 55,

*PostVisit*: 58, 53, 48, 52, 5, 19, 23, 25, 27, 31, 35, 51, 39, 41, 10, 43, 12, 34, 14, 9, 45, 16, 22, 18, 30, 47, 38, 62, 61, 57, 56, 59, 60, 1, 54, 55, 2, 49, 50, 3, 46, 44, 42, 40, 36, 37, 32, 33, 28, 29, 26, 24, 20, 21, 6, 17, 15, 13, 11, 7, 8, 4

*Sorted PostVisit*: 62, 61, 58, 57, 56, 53, 52, 51, 48, 47, 45, 43, 41, 39, 38, 35, 34, 31, 30, 27, 25, 23, 22, 19, 18, 16, 14, 12, 10, 9, 5,

*Topological order*: [MATH 1390, MATH 1591, CSCI 1470, MATH 2330, MATH 3320, CSCI 1480, CSCI 2440, CSCI 3380, CSCI 2320, CSCI 4390, CSCI 4350, CSCI 4310, CSCI 3385, CSCI 3381, CSCI 4490, CSCI 3370, CSCI 4320, CSCI 3360, CSCI 4370, CSCI 3350, CSCI 3345, CSCI 3335, CSCI 4355, CSCI 3330, CSCI 4365, CSCI 4353, CSCI 4340, CSCI 4315, CSCI 4300, CSCI 4345, CSCI 3190]

***Screenshot of results***:

**Conclusion**

In conclusion, we find out that DFS algorithms is a very useful tool for scheduling things like classes from given dependencies like its prerequisites. In computer science, applications of this type arise in instruction scheduling, ordering of formula cell evaluation when recomputing formula values in spreadsheets, logic synthesis, determining the order of compilation tasks to perform in make files, data serialization, and resolving symbol dependencies in linkers.

In this project, we simplified and solved a real-world problem by applying an existing algorithm into a real-world application. We designed reasonable software solutions, implemented and verified potential solutions to them. Team work was a very important aspect of our project because we found that work like this is better split among members. Individually, we couldn’t exactly figure it out but brainstorming as a group gave us an edge to solving the problem.

There were a lot of considerations and trade-offs we made as a team, most important of all was the language we wrote the solution in. We used C++ in the last project because all our team members understood it but for this project we changed to Java. The majority of team members understood the language and it offered better functionality and ease of use that C++ didn’t.

In the future, we would consider using BFS in order to find the shortest path if one exists because it is a lot simpler and doesn't need any data structures. We can just keep a tree (the breadth first search tree), a list of nodes to be added to the tree, and markings (Boolean variables) on the vertices to tell whether they are in the tree or list.