Programming Challenge 07 Description and Testing

Design Approach

When approaching this problem, I was first drawn to the picture that Professor Morrison included in the Canvas Page. Once I saw the graph that Professor Morrison drew, I thought about using a form of Djikstra’s algorithm to find the correct path for the password. To create this program, I rewrote Professor Morrison’s graph code to create my own graph class with it’s own function to find the password.

In the graph class, I use an adjacency list to store all of the vertices and the edges. In the code, it is actually stored as a VECTOR to a Vertex class which holds all of the edges inside of the Vertex. There is a function that uses a mixture of a depth first search alongside a weight edges algorithm to find the longest path between two possible nodes. This algorithm, which is inside the graph class, takes in an origin node and a destination node. The algorithm uses a stack, a vector of parents, and a vector of weights. With the original node, add each of the outgoing edges destination nodes to the stack, and update the weights and the parents inside of the respective vector. Then get the top node from the stack, and perform the same operation. Add each of its outgoing edge destinations to the stack, and update the parents and weights nodes. However, only update those nodes if the weight to the current node plus the weight to the destination is greater than the weight already at that destination. This will ensure that it will find the path with the greatest weight. This algorithm will work well with finding the longest path between two nodes in this problem. However, this algorithm will lead to an endless loop if any loops are present in the path of the graph. The algorithm does not terminate when it reaches the end node, it will continue to search the top nodes off the stack until the stack is empty. This is an exhaustive search and will guarantee that the longest path between two nodes is found. But this will not prevent the algorithm from going between two nodes in an endless loop if it exists, so this algorithm will fail with loops present. In this problem, no loops would ever be present in the pins so this issue is of no worries.

In my passcode.cpp file, I have four functions. The first function will read in the pins from a file and store the contents into a character vector. The second function will read in those pins from a character vector into a character graph. The third function will then return the correct password from the graph. The last function will write the password out into a file and save the file.

Testing Approach

In order to test the functionality of my program, I created a python script and a bash script to test if the PC07 is working properly. The python script will create a randomly generated password with a length of ‘n’ which is passed at the command line, and then it will generate pins to make that password solvable and save those pins into a test file for PC07 to read. The python script will also save the password into a different file to test against with PC07. The bash script will run both the python script to generate the different passwords and then it will run the C++ script to test if PC07 is working properly.

PYTHON:

In order to create proper test scripts, I first needed to generate a valid password. In my Python script, I have a function which contains a list of all the integers as characters. I then have a for loop that will randomly choose an integer character, add it to a string, and then pop it from the list. The for loop will run for ‘n’ times, and return a string that is a valid password with uniquely randomly ordered integers.

Then to create the pins, I have a dictionary that will first store every 2-pair adjacent integers in the password. For example, if the password was ‘123’, then the dictionary would hold ‘12’ and ‘23’. This is necessary because you need to have every pairing in at least one of the pins, otherwise the pins will have ambiguity when trying to recreate the password. For example, if the password is ‘1234’, then having pins ‘123’ and ‘124’ leads to an ambiguous situation where the password could either be ‘1234’ or ‘1243’. So it is necessary for the pairing ‘34’ to show up in a pin to resolve the ambiguity of the password. This can be applied to all pairings, so there must be at least one pin for each digit pairing. After creating the dictionary, the program will continue to create random pins until a valid path to create the password is generated. This is kept track of using a counter to see if all valid pairings have been spotted. To create random pins, I used a for loop that will generate a number between a given range so that only valid pins will be generated. The first digit added to the pin can be the first digit in the password or the third to last digit in the password. It cannot be the last digit or second to last digit, otherwise no valid pin would be possible to make. The second digit added to the pin must be greater than the previous one, and it also cannot be the last digit for the same reason. And the third digit added must be greater than the second digit added. This ensures that only valid pins are generated. Once the valid pins are generated, they are then checked to see if they contain a valid pairing of numbers, such as ‘12’ when mentioned in the passwords earlier. If the pins do not contain any new pairings, it is ignored. Otherwise, the program will increment the counter and then add that pin to a list of pins. Once every pairing is found in pins, the set of valid pins to generate the random password is generated. These pins will only correspond to the random password that was given.

The python script will then save the pins to a text file and save the password to a different text file.

BASH:

The bash script will first use the aforementioned Python script to generate the 5 test scripts, then it will make the executable for PC07. Then it will create new text files containing the password generated from PC07 after reading in the pins file. The bash script will use diff to test if the PC07 has the same results as the password generated from the Python script. If it does, then it will remove both of those text files and continue to the next test. If the password from PC07 is equivalent to the password from the python script, then we know that PC07 works because it generates the correct password when given a set of pins for a valid password.

To test PC07, use PC07test.sh to generate random passwords and corresponding random pins, and then have PC07 generate a new, equivalent password from the set of pins.