CS253-Assignment 3- Soham Ghosal, Roll:180771

[Metric used for comparison: Branch Coverage]
Please Note: I tried tinkering around with other codes for `p.cpp`, and most of them worked in a lesser amount of time than the code with which I am submitting it. So, the time taken by my code is probably close to the worst case time. Replacing `p.cpp` with some other code will most likely give faster results.

Scripts:-

1. generate.sh
2. reduce.sh

Detailed explanation of each script:--

generate.sh

This script primarily does 3 things.

1. Clean up all the leftover stray files, before the current execution.
2. Generate n test cases, ie. 2n integers, where n is an input by the user.
3. Get the coverage data using gcov, extract and store the necessary information, which will be required later by the "`logic.cpp`" file during the reduction algorithm.

-->How did I generate random numbers?

Since `$RANDOM` in bash can only generate random integers between 0 and 32767,which was not sufficient for this purpose, I decided to split my required range into three parts. The value of `INT_MAX is 2147483647`, and that of `INT_MIN is -2147483648`. I divided this into groups of 4-3-3, that is, `2147483647` was split into `2147-483-647`. Then I could easily generate 3 random numbers for this, the first one less than 2148, the second one less than 484 and the last one less than 648. For negative numbers, I decided to generate another number between 0 and 1. If this number was 0, the generated number would be treated as negative, otherwise positive. The syntax to generate random numbers between 0 and B is:

```
$RANDOM % B
```

---> Running gcov on `p.cpp`

The first step was compiling the code, using the g++ compiler. I also used the `-fprofile-arcs` and `-ftest-coverage` flags, as mentioned in class. Then I sent the required inputs [a and b] to the binary executable created in the above step. I also redirected its output to /dev/null to silence it.Then I used the gcov command with -b and flags to get the coverage data for the current test case.
[Motive behind this]: For every test case, I needed the control flow, so that I could deduce which branches was visited by that test case. Since I wanted this data independent of other test cases, I had to delete the `.gcov,.gcno, and .gcda` files created after every test case, since otherwise, the results would have been accumulated.

I used the grep command, to extract all strings matching "branch" from the p.cpp.gcov file. Then I checked the 4th word of all these lines, if this word was 100%, the .gcov file said taken 100%, which according to me, means that all the test cases which reached the if/else condition, took this branch. I ran a lot of test cases individually, and based on the .gcov file and the inputs, this was what I deduced. Anyway, if the 4th word was 100%, I sent in a 1 in `new.txt`, otherwise a 0. So, `generate.sh` also creates a 2D matrix in `new.txt`, where if the ij-th entry is 1, it means that the the control fell into the jth branch, during the execution of the ith test case.
Finally, I also ran `gcov` for one last time, after running the executable on all the n test cases. This gave be the overall branch coverage, which I redirected to another file-- the input for `logic.cpp` which implements the main reduce logic.

reduce.sh

The initial part of this script is just to handle the scenario when the user generates the inputs only once, but wants to run the reduce script multiple times, possibly with different values of k. Here I read k, and then compile the `logic.cpp` code, which implements the actual reduce logic. `logic.cpp` basically gives me the test case numbers which should be selected for maximal coverage. I redirect this to a file `choice.txt`. I use a bash array to store these relevant test numbers.
Then, I need the updated value of the branch coverage, along with these tests only. So I recompile the `p.cpp` code, and this time only run the binary along with the relevant test cases. The output is redirected to `results.txt` along with the necessary info.

Logic Behind the Reduce Algorithm{Logic.cpp}

The `generate.sh` creates a n*b matrix, where n is the number of test cases and b is the number of branches. This matrix has 0 and 1 as its elements, as described above.
For each test case, I wanted to store the branches through which it passed. To do that I created a 2-D vector `hold` where hold[i] will have those branch numbers which were taken during the execution of the ith test case.
Then the problem reduces to the maximal coverage problem.
Quoting from Wikipedia,

```
As input you are given several sets and a number k. The sets may have some
elements in common. You must select at most k of these sets such that the
maximum number of elements are covered, i.e. the union of the selected sets
has maximal size.
The maximum coverage problem is NP-hard, and cannot be approximated within
1-(1/e)+o(1)= 0.632 under standard assumptions. This result essentially
matches the approximation ratio achieved by the generic greedy algorithm
used for maximization of submodular functions with a cardinality
constraint.
```

As mentioned here,this is a NP-Hard problem, that is no polynomial time solution exists as of now. Thus greedy is the way to go.

```
The greedy algorithm for maximum coverage chooses sets according to one
rule: at each stage, choose a set which contains the largest number of
uncovered elements. It can be shown that this algorithm achieves an
approximation ratio of 1-(1/e)
```

This is exactly the algorithm which I implemented. For each `available` test case(test case which hasn't already been selected), I maintain a count of the number of branches in its domain which are not visited. Then I choose the maximal such set, ie. the test case which has the maximum number of non visited bramches among the set of available test cases.

```
for(int itr=1;itr<=k;itr++){
        int maxx=-1;int id=-1;
        for(int i=0;i<n;i++){
            int cnt=0;
            if(available[i]){
                for(auto x:hold[i]){
                    if(!visited[x]) cnt++;
                }
            }
            else continue;
            if(cnt>maxx){
                maxx=cnt;id=i;
            }
        }
        available[id]=false;
        for(auto x:hold[id])
            visited[x]=true;
        taken.push_back(id+1);
    }
```

The final result is stored in the `taken` vector, which gives us the k required test case numbers.

```
Time Complexity: O(n*b*k), where n=number of test cases, b=number of
branches, k=required set size of the reduced test suite.
```

Purpose of the other helper files:

1. `bin/p` : binary executable for p.cpp
2. `data.txt`: Prints out the coverage data after every test case. The first column is the Line Coverage and the second column is the Branch Coverage.
3. `new.txt`: The matrix created by the `generate.sh` file is redirected here, and also `logic.cpp` takes its inputs from this file.
4. `results.txt`: The final results are summarised here. I have compared the branch coverage for the entire test suite with the branch coverage for the reduced test suite.

How to Run?

1. After unzipping the folder, just run the following:

```
bash generate.sh
```

Enter the value of n as required.
2) Then run:

```
bash reduce.sh
```

Enter the value of k as required.
3) The results should be ready in results.txt