# Part2 – test generation algorithm

Yi-Chiao, Yang
932526065

In my algorithm, I would like to find some paths that have coverage less than median coverage through the breadth fast search. After I find these paths, I will focus on these paths and their information for testers to improve their programs. In CS562 class, we have learned how to design a tstl file of test software for python library to find bugs. We might have good coverage on the test software, but not all functions on test software we visit the same time. To explain, on some functions we visit them many times, but on other functions, we might visit them less time. In order to solve this embarrassing situation, I would like to design an algorithm to provide useful information for testers to enhance their test software.

It is important to make sure no mistakes on missing paths. We want to visit these paths whatever it has great coverage or fewer coverage, we do not like to miss on of them. Thus, we need an algorithm to search every possible path. For this idea, I find that the breath fast search algorithm is a good method to help me process it. Breadth fast search algorithm starts a node to visit following its adjacent neighbors, and visits these unvisited adjacent neighbors until the node visits all of nodes in the pattern.

There are two sections in my procedure, the first section is to find bugs in test software, calculate which path has less coverage than median coverage. The second section is to improve chances on visiting these paths more times. We set a time for running the test generation, so we need to consider the partition of used time. We might have lots of unvisited nodes need to visit them. If we set less time on visiting these nodes, we might not have enough time to visit all of unvisited nodes. If we set less time on improving coverage of paths, it might hardly enhance the efficiency on test generations.

```
Used_time_1 – start_time_1 < estimation_time /2
Procedure test_generation( node x)
        Create a queue Q ← states in sut
        Make x as visited
        Enqueue x onto queue Q
        While x in the queue Q{
                Find_bugs_via_enable( )
                Dequeue a vertex from Q into v
                For each w adjacent to v {
                        If w unvisited{
                        Make w as visited
                        Enqueue w onto queue Q
                        Collect_coverage( )
                        }
                }
                Collect_coverage( )
```

}
Used_time_2 – start_time_2 < estimation_time/2
Improvement_coverage()
Print_nformation()

There are several functions I have not finished yet, such as Find_bugs_*via*_enable( ), Collect_coverage( ), Improvement_coverage() and Print_nformation()

Find_bugs_*via*_enable( ): find bugs through enable() function
Collect_coverage( ): count coverage, and store these information
Improvement_coverage(): finds some paths that have lower coverage and make a connection or a few connections for them.
Print_nformation(): I am not sure this function how to do, it is still my plan. I want to print some useful information about these paths that have fewer visitors.

In my future works, I would like to finish these functions, and make the logic and structure stronger.