1. How can graph-based testing be integrated into existing software development and testing processes? (e.g., *who should design, write, and execute test cases? when in the process? how to document and save the test cases? how to document and save the testing results?*).
2. What are the limitations and potential biases associated with existing graph testing tools and frameworks?

Graph-based testing can be integrated into an existing software development or test process like any new framework would. First, a few roles would probably need to be created (or assigned in addition to existing responsibilities) in order to facilitate appropriate implementation and sustainment of the process. Some key roles that I can think of are designers, engineers to automate the tests, and test executors. This will ensure someone is solely designated to developing a comprehensive set of flow control diagrams, making sure all possible conditions are met. Another individual would be responsible for automating the tests, making sure each test is correctly implemented. Someone else would execute and document the test cases, ensuring that each test functions as expected and follows the flow control diagram created by the designer. Having each role separated out will allow for three separate pairs of eyes to look at the testing and identify possible issues or lack of coverage.

Inserting this would definitely be development process dependent, but let’s take a TDD approach as an example. A flow control diagram to represent the test could be created. Then the developer would develop based on the test and associated diagram. Then an automation engineer would automate the test. Finally, the test executor would evaluate the test to verify that it meets the original definition of the test as defined by the flow control diagram. The test executor’s role could be easily improved by putting the automated tests in CI/CD pipeline, thus only leaving evaluation as a task. The automated tests would then be saved with the source code and results could be saved and maintained by the executor.

One potential limitation that I can think of for existing graph testing tools or frameworks is the complexity of the tool. If the learning curve is too high, it may make it difficult for test engineers to adopt the tool. On the other hand, if the tool is automated to some degree and easy to use (analyzes source and generates a graph), it may not be supported by a lot of languages. At the end of the day, a human is in control of the testing, so there will be human bias. No matter how good a tool or framework is, it could still become less effective due to human bias.

**I personally believe that graph-based testing would be best used to help with test coverage. Because of that, I feel like tests should be ran on obvious areas where issues may occur and once all those tests pass, graph-based testing should be used to ensure test coverage is good. If you find an issue before using graph-based testing, you can just fix it and not have to worry about updating the graph you just made (if the fix requires an update to the graph). With that in mind, I feel like testers/QA engineers should be the ones mainly using graph-based testing. This is not only because I believe graph-based testing should be done at a later stage, but it is also because I feel like the developers' time could be better spent on other activities other then creating a graph for testing. This kind of links back to my post from the previous forum, where I think developers should run some tests initially to hopefully catch most of the more impactful/obvious bugs, and then hand it over to QA for more in-depth testing.**

**A limitation that could come up is the fact that if a piece of the graph was missed (not input by the human) then the program probably won't be smart enough to recognize that. There's still a human element involved, and even if the tool automatically made a graph just based off the code, there would probably still be double checking. Also, there may be some tools with a limit on how many nodes and edges the graph can have.**

Hey Steven,

I mentioned in my post that when to integrate graph-based testing is largely dependent on the existing development approach or process. I agree with your defense for why graph-based testing should take place later in the testing process, but I would like to suggest that it can also take place in the beginning. For example, in TDD you design the tests first and then develop around those tests. Developing the graph during this stage may help the developer visual the logic better and increase time spent developing. There is probably an argument to be made on if that time should be spent creating graphs or developing, but that is subjective and will depend on who you ask. That’s why I don’t think there is one right answer. I think both of our approaches make sense when deciding when to implement the process. I would also like to briefly touch add to some of your potential limitations of graphing tools. Some of the automated ones are language specific, so if your application isn’t one of those languages, then you would be able to use automated tools. It sounds like this type of approach is super effective in a few, niche cases.

**Even in a project that already has software development and testing processes in place, adding graph based testing could provide value to the project. Creating a graphical representation of the code could be a final step in the development process, allowing developers to have a better understanding of the organizational structure of the code and the overall logic. This would allow for better visibility of potential weak points that could be communicated to the test team to focus their efforts towards. From there it would likely be up to the test team to determine which paths/nodes to test and the level of coverage desired. Including this graph and the selected test paths as an artifact would be helpful to future developers on the project to have a better understanding of the code at hand.**

**One bias of existing graph testing tools could be that only the path of the code is considered during testing, and other key factors such as processing time or input variability may fall to the wayside. I think it's important to look at graph testing as a tool to help create better test cases and not as a standalone solution for all testing needs of any project.**

Hey Ryann,

I definitely agree with you that graph based testing could be introduced at the end of a development process. It still adds value for the developer to see a bigger picture of the code structure as well as help the testing team create better tests. I also think there is value in inserting graph based testing in the beginning of the process as well. Creating the graph to represent the code before developing may help identify glaring logic issues before development starts. It also may help the developer understand how the current piece of code will fit into the larger system. Truthfully, I think an argument can be made for inserting graph-based testing at any stage of the process. Practically, it will depend on the project, the people working the project, and the people leading the project. Since its integration into a project can be more person dependent than it is defined by a set of guidelines, it leaves the door open for human bias as well, assuming the graph is not generated based on code.