

Assignment 3 Report

Rayan Nasrallah

1) Describe how the elements covered in weeks five to twelve (lectures and tutorials) change your vision on the code you delivered.

I have come to realize the extent to which my code violated fundamental software engineering principles. In my code almost all the programs did not follow the SOLID principles, mostly in the Maze module. The maze solver module solves a path, verifies the path, and validates moves, making it very hard for the program to be open to addition of, for example, another algorithm. The module is also not easily extendable without modifying the existing code. Moreover, Liskov's Substitution Principle and the Interface Segregation Principle are not applied at all, making it so that clients are forced to depend on methods in the module that they might not use. No interface was used for the solver and the solver module had a lot of technical debt, making the program closed for addition and open for modification, and making it extremely difficult to add another maze solving algorithm. I could have easily applied GoF and GRASP patterns such as the strategy pattern to the maze solver module, which would have facilitated extension of the code.

2) What do you think of the grade you received at that time? If you'd deliver this code now, do you think your grade would stay the same, as the grader will assume you now know more than earlier?

I think that the grade I received was fair and reflected the structure of my program accurately. I was unaware of how to apply SOLID principles, OO design, and abstractions, which was expected of me at the time. I failed these three sections in assignment one, however they were not worth a lot of marks as we were not required to have an excellent understanding of how to apply them. Now, it would be expected that I refined my understanding in these three areas to the degree that they should now be easy to apply when writing code, hence I think they would be worth more of my grade. Thus, if I couldn't now apply these principles and delivered the same code, failing in these three sections would cause me to receive a lower grade. Moreover, at the time of assignment one, it wasn't expected of us to have any information on GoF and GRASP patterns and how to apply them, and since none were applied to my code in assignment one, the lack of patterns would contribute to decreasing my grade even further.

3) What part of your code is in the most debt?

The part of my code that is most in debt would be the module for the right-hand maze solver, the Maze module. The Maze module had a lot of hardcoded logic, especially the movement logic. For both the 'findPath' and 'checkPath' methods, the movement logic and direction handling are hardcoded, and both methods also contain a significant amount of duplicated logic. Furthermore, in the right-hand solver I used very long conditionals, many Booleans, and had a lack of helper functions, making it extremely difficult to modify the code if need be. Both these aspects make the code less flexible and harder to maintain. There is also a lack of abstraction and error handling within the module. For example, there is nothing to check for if the path is out of bounds in the maze array, and the use of a 2D array to represent the maze itself and no use of an interface or abstractions for the module create low cohesion and tight coupling.

4) How can one formalize this maze exploration problem as a graph one?

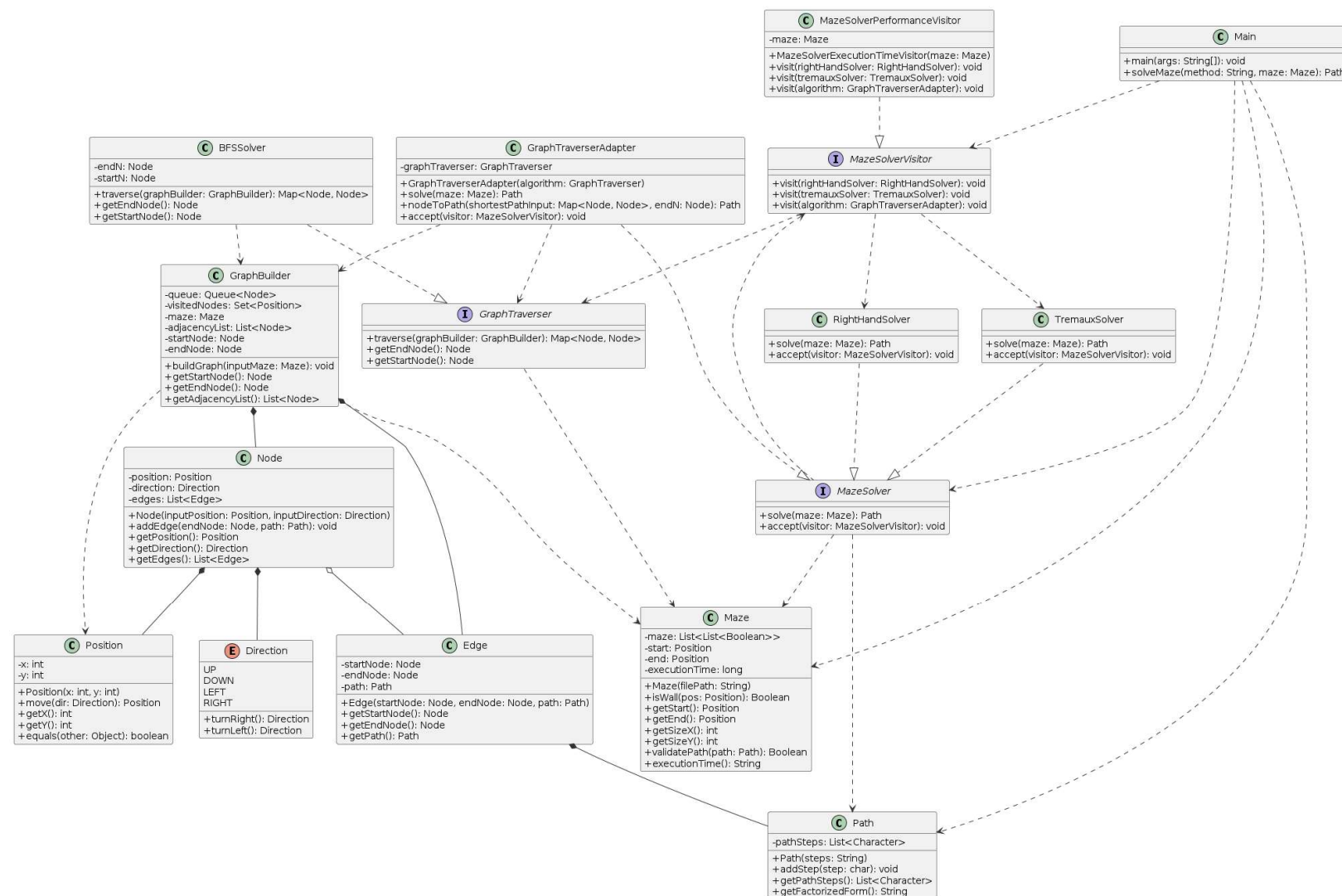
To formalize this maze exploration problem as a graph one, we must find equivalences in the maze that can represent different parts of a graph problem. The graph representation would be a graph of nodes and edges. This is because we can navigate through the maze (i.e. the graph) based on the validity of moving in a certain direction or moving to a given position/node (based on whether the position is a wall, or out of bounds). First, we can represent the start and end nodes as the entry and exit points of the maze, and our goal to be finding the shortest path from the start node (entry point) to the end node (exit point). We can represent each position (e.g coordinate) and direction pair in the graph as a node in the graph, and all moves adjacent to a given node as the edges, or in other words and

possible/valid moves represented by the sequence of instructions consisting of 'F', 'FR', or 'FL' required to move from the one cell (node) to another as the edges. To solve the graph problem, we can implement an algorithm that finds the shortest path from the start node to end node.

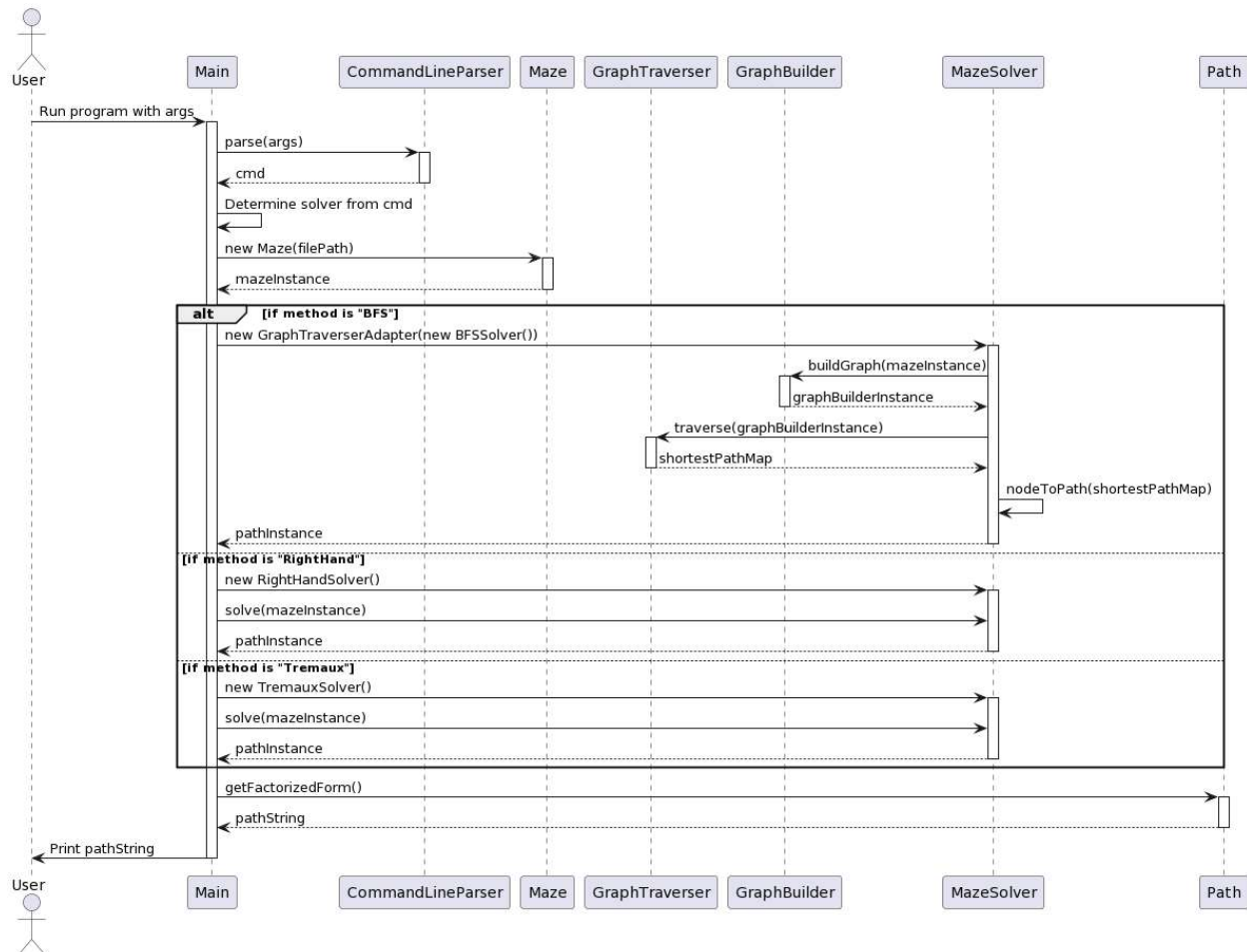
5) Which kind of algorithms can be used to support such exploration on top of a graph?

Once the graph is built, a range of graph traversal algorithms can be applied to find a path from the entry node to the exit node. The algorithms explore the graph, starting from the entry node, and follow the edges to navigate through the maze. Any graph traversal algorithm implements the 'GraphTraverser' interface, which requires the path to be returned as Map<Node, Node>, which is then reconstructed based on the sequence of nodes visited by the algorithm to represent the shortest path as type Path. For the maze exploration, I have chosen to use a breadth first search (BFS) algorithm to support such exploration on top of a graph. A first in first out (FIFO) queue will be used to store nodes that are explored, ensuring that nodes are explored in the order that they are discovered, and hence facilitating level by level exploration of the graph, or in other words, newly discovered nodes will only be explored when all other nodes at the same level have been explored. Nodes are enqueued to the queue when they are discovered, and dequeued when they are explored, hence guaranteeing that every node is only examined once, and ensuring that no node is visited more than once in a single possible path. All other graph traversal algorithms would similarly traverse the maze, but by using a different strategy.

6) Provide a high-level class diagram describing these abstractions.



7) Using a high-level sequence diagram, show how your objects interact, from parsing the command line to printing the result.



8) How do you support the evolution of this Maze Runner in the future? Identify what can vary in this product and to what extent you are (or not) “protected from variations”.

By implementing the strategy pattern through the ‘GraphTraverser’ and ‘MazeSolver’ interfaces, adding new solving algorithms, whether they require the use of a graph representation of the maze (e.g. BFS, DFS, Dijkstra etc.) or the use of a regular maze representation (e.g. Trumeaux, right hand, etc.) is made very easy, and hence supports the evolution of this Maze Runner in the future. The use of the strategy pattern and application of the interface segregation principle ensured that the program was decoupled, easy to understand, and easy to extend, hence facilitating the ability to add another ‘algorithm’ (extend the program) without needing to modify any of the program. To implement a new algorithm, it is as simple as creating a module for the algorithm that implements either the ‘GraphTraverser’ or the ‘MazeSolver’ interfaces depending on what Maze representation the algorithm uses. Furthermore, if the client wishes to measure the performance of each module differently, or add new operations to the available algorithms, the client can do this by creating another module that implements the MazeSolverVisitor interface, instead of modifying the code for any of the algorithms, or any of the code at all. This is facilitated by the application of the visitor pattern via ‘MazeSolverVisitor’ and ‘MazeSolverPerformanceVisitor’.

Moreover, because the provided Maze module encapsulates the maze structure, it becomes very simple for the client to introduce a new maze representation without having to deal with reading in the maze file, but rather only using the Maze module representation of the maze. The ‘GraphBuilder’ module in my code converts the maze representation of the Maze module representation into a graph composed of ‘Node’ and ‘Edge’ objects, without

having to read the text file. This adds an extra layer of encapsulation between the maze representation and the graph traversal algorithms, allowing for the implementation of any graph traversing algorithm to find the shortest path in the maze without having to modify the existing maze representations, add a new one or have to modify any of the code at all, as the algorithms can operate on the graph representation without needing to understand the underlying maze representation. By handling the conversion of the maze to a graph, the 'GraphBuilder' module allows for reusability and follows the open closed principle, hence promoting the evolution of the maze runner program by facilitating the ability to implement a large range of graph traversal and maze solving algorithms. Because of this, and due to the implementation of the strategy and adapter patterns, the program is well protected against variations in graph traversal and maze solving algorithms. In terms of the maze representation for the maze solving or graph traversal algorithms, I am not well protected from variation. This is because each interface must implement a specific representation of the maze to find a path, and if the structure of the maze representation for either interface changes, this will cause the algorithms to stop working. To use a different maze representation and implement the same algorithms, one must recreate the algorithms. For measuring performance, I am well protected from variation. This is due to the implementation of the visitor pattern, which allows for the measurement of performance without needing to alter any of the code in the algorithm modules, hence allowing for a variety of performance measurements by only changing the methods in the 'MazeSolverPerformanceVisitor' module or by making another module that implements the 'MazeSolverVisitor'. Overall, the Maze Runner program is open to variations in solving algorithms and measuring performance, as it can be extended to implement a range of algorithms and measure their performance without needing to modify the existing code. However, changes to the maze representation, if needed, will require extensive modification of the existing code.

9) If you have used some GoF or GRASP patterns: Which ones? Where? Why?

In the provided solution code, the strategy pattern was initially applied to the 'MazeSolver' interface, which algorithms that solve the maze using the Maze representation use. I also decided to apply the strategy pattern to the 'GraphTraverser' interface, to ensure that algorithms that use the Graph representation are separate from the others and to allow the graph traversal algorithms to take in a different input and produce a different output. The use of the strategy pattern allows the selection of different graph traversal algorithms at runtime and allows for the extension of the program (i.e. the implementation of another algorithm) without needing to modify any of the existing code. All that is needed to be done to implement another algorithm is to create another module that implements either the 'GraphTraverser' or 'MazeSolver' interfaces, depending on what maze representation it uses, and add the algorithm as an option in the 'solveMaze' in main. I also used the adapter pattern to make the program closed for modification and open for extension. The adapter pattern is used to adapt the 'GraphTraverser' interface to the 'MazeSolver' interface, hence allowing for any graph traversal algorithm to be used as a maze solving algorithm, and consequently making the implementation of algorithms easier and more flexible. Using the adapter pattern also facilitates the conversion of the maze representation in the Main module to a Graph representation which can be used by the graph traversal algorithm, and converts the Map<Node, Node> return type of the graph traversal algorithm to a Path object, which is compatible with the Main module. Overall, it facilitates for the easy addition of a graph traversal algorithm without needing to modify or worry about how the underlying logic to convert data works. The last pattern I used was the visitor pattern, which was implemented through the 'MazeSolverVisitor' interface and 'MazeSolverPerformanceVisitor' module. The visitor pattern is used to separate algorithm execution from the object on which it operates, which in my program allows for different 'MazeSolver' or 'GraphTraverser' implementations to be visited to have their execution time, or any other metric, measured without having to modify any of the graph traverser or maze solver modules. All three patterns play a role in facilitating the ability to extend functionality without modifying the existing code.

10) One must not test 100% of their product but focus on what is relevant. How did you identify what to test? To what extent do your tests give you confidence in your code?

To identify what to test, I thought about what components of my program were fundamental for the Maze Runner programs overall functionality, correctness, and reliability. I tested the primary functionalities of the program, for example the graph traversing BFS algorithm, the forming of a graph which a graph traversal algorithm can use, and the ability to adapt the graph traversal algorithm to allow it to behave like a maze solving algorithm, which are all primary components of the Maze Runner program and its ability to implement graph traversing algorithms. Certain modules in the program such as the 'GraphTraverserAdapter' and 'GraphBuilder' modules which contained complex logic were prioritized when testing, as the logic in these modules needed to be tested to ensure that they can accurately handle various scenarios and edge cases. Moreover, I chose to also test data transformation processes, particularly the ability of the 'GraphTraverserAdapter' to convert the return type of a 'GraphTraverser' algorithm to a Path object, as the overall functionality of the program and other modules was dependant on their accuracy and consistency. I tested:

- The ability of the adapter to reconstruct the path from a graph traversal algorithm.
- The ability of the 'GraphBuilder' module to correctly build a graph consisting of valid nodes and edges by traversing the Maze module representation of the maze.
- If the graph representation correctly represents the structure of the maze and can be traversed,
- The functionality of connected nodes and edges, and the ability to extract key information from these nodes and edges.
- The 'BFSSolver' module to determine if it was able to produce a correct and shortest path of a maze by traversing the graph representation of the maze.

The tests that were conducted give me a significant level of confidence in my code. The tests focus on core components that the program relies on to function accurately, such as tests done to validate the correctness of the graph construction, graph traversal algorithm adapter, and the graph traversal algorithm itself (the BFS algorithm). All the tests executed as expected, hence demonstrating that the primary components that are needed to implement any graph traversal algorithm, function as required and support the addition of any number of maze solving and graph traversal algorithms in the future. The successful execution of the tests also demonstrates the ability of the program to handle any number of scenarios (e.g. different mazes) and edge cases and demonstrates the ability of the program to facilitate extension and support any graph traversal or maze solving algorithms.

11) What other kind of metrics could be helpful for this product? Any idea how to measure them?

There are two ways to measure an algorithms performance; its space and time complexity. And since we are already measuring time, another metric that could be helpful for the product could be measuring memory usage (i.e. space). For large mazes, algorithms that utilize less memory are important. An algorithms memory usage determines if an algorithm can operate on very large mazes or complicated mazes effectively and efficiently. Efficient memory usage is also important to prevent memory overflow or system crashes and makes the algorithm more scalable (able to handle large and complex mazes without being limited by memory related constraints. Memory usage also impacts the performance of an algorithm. Algorithms that use less memory has a lower complexity and thus a lower execution time, therefore, optimizing memory usage results in a faster algorithm. Overall, having memory usage as a metric allows for the comparison of the efficiency of different algorithms; an algorithm that utilizes less memory is more efficient. To measure memory usage, you can use the Runtime module provided by Java's libraries to retrieve information about the Java Virtual Machine memory usage. Another metric that could be helpful for this product, is the optimality of an algorithm. The optimality of a graph traversal algorithm can be measured by comparing the path constructed by an algorithm to the shortest path of a given maze, which can be calculated like Speedup. Algorithms that don't necessarily find the shortest path, like DFS or the right-hand algorithm, can be measured for how optimal they are, thus allowing for a comparison in performance of algorithms that don't find the shortest path.