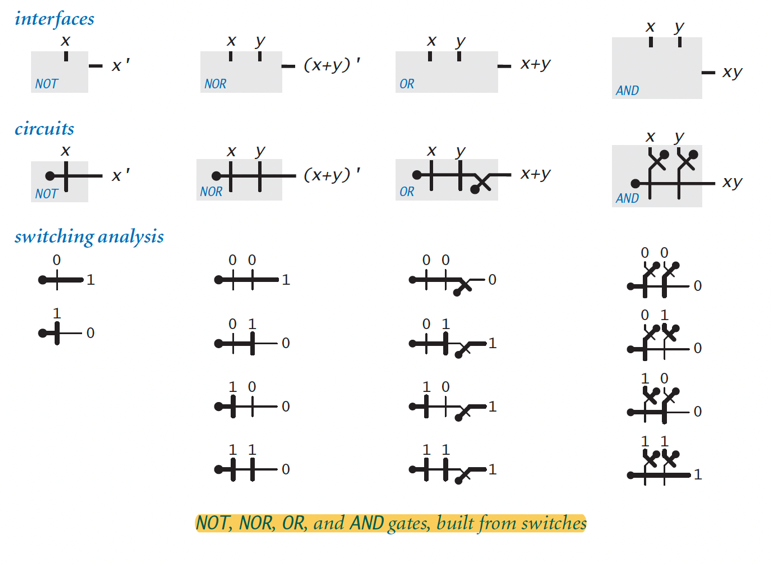
**Circuit Board**

**Introduction**

           Circuit design is not a field I have experience with. However, the CPU structure was something I had learned before. It was a topic that piqued my interest and was something I was willing to learn more about. Computer organization emphasizes the importance of maximizing efficiency and reducing cycle time. One significant part of efficiency is effectively implementing the caches and memory; the other is circuit design. For modules to communicate data, buses or wires must connect computer components to deliver data in a sequence of bits. I want to write a program that randomly generates a pattern that mimics specific circuit patterns using iteration and repetition to express my interest in circuit design.

**Research**

With smaller computer chips, buses are made to deliver data in the fastest way possible to reduce the time of instruction executions.[[1]](#footnote-1) In order words, the shortest paths connecting a source to its designated ports must be found. There are many algorithms to choose from; shortest path algorithms are a topic that scholars extensively study. Many complications may arise in real life designing a circuit; for example, the material used for wires may differ, in which each edge connecting ports may have a different "weight," signifying the difference in priority when selecting a path.[[2]](#footnote-2) In particular, single source algorithms such as Dijkstra's single source shortest path algorithm may work very well in designing circuits for CPU in real life. However, to simplify this project's problem, I will assume all weights on edges are equivalent.

It is also important to note that wires are not used only to transport bits; they are also responsible for creating logic gates and switches, which are used extensively in multiplexer and calculation units such as the CPU's Arithmetic Logic Unit (ALU).[[3]](#footnote-3) Intersecting input wires are used as NOT gates, which form the basics of the rest of the combinational circuits, logic gates for calculations. Most times, the implementation of these gates is hidden in circuit designs to improve readability. I do, however, will include resemblance to the inner structure of the logic gates by allowing wires to intersect one another in specific angles.

“Not, NOR, OR, AND gates, built from switches.” Basic Logic Gates. (Sedgewick & Wayne, Computer science: An interdisciplinary approach 2016)

**Design**

           The project has two versions. Please look at CircuitBoardv2 for the final product.

Data Structure:

The data structure was an essential part of the implementation of this project because it determines how paths are grouped underneath the code.

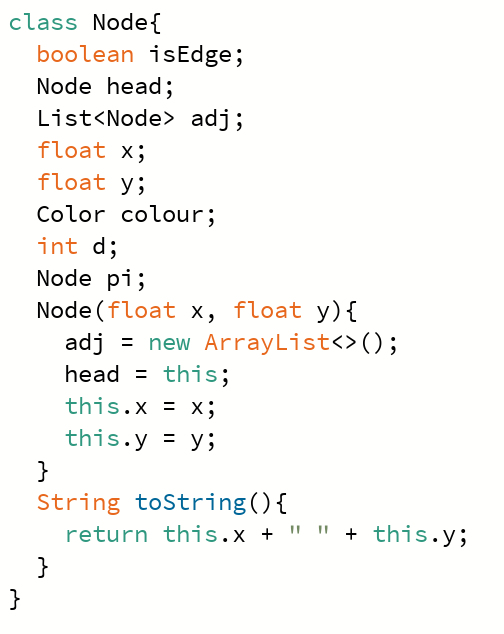
As the circuit board can be seen as a board with ports in specific coordinates, I used a 2D array that stores nodes.

Figure Node Class from Code

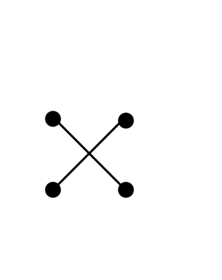
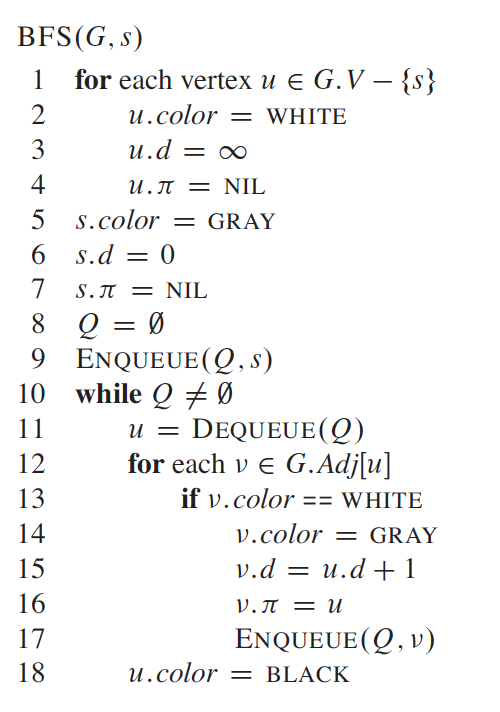
For this project, I used Union and disjoint sets with nodes that indicate ports and paths in the circuit. The Node class provides a blueprint that stores information on each element in the board. Information that would determine whether a point is a port or a regular Node that can be visited in a path, as well as information that will be useful in shortest-path algorithms. Paths between the source and the port are essentially the connection of nodes that lead the source to the port, and this connection is made possible with Union. The Union class contains methods that "unionize" two nodes, given that it passes some conditions. Unionizing nodes allowed me to create disjoint sets among the nodes in the board, which further allowed me to limit nodes of disjoint sets from accessing nodes in each other's paths. It should be noted that this limitation does not prevent two paths from crossing. As shown in the figure 2 on the right, the two paths may cross without using nodes in either set.

Figure Intersecting Paths

Module Representation:

           Circuit boards would only be meaningful if they serve the purpose of connecting modules in a CPU. Modules are represented as square blocks of an empty center on the board, i.e., null instead of being nodes. This feature is to prevent nodes from creating a path through a module. Each module (except the center one) has its size randomly generated. The ports of each module lying on the borders of the module are marked as edge using a Boolean instance in the Nodes class, meaning the edge/sides of a module. The center module of the board has its borders outlined, and that unit resembles the control unit in a CPU. The ports on the control unit are implemented slightly differently from the ports on the rest of the modules. The control unit ports are marked as start edges, meaning all sources will come from these ports, and these control ports are where most of the wires drawn/originate from.

Path Finding:

           I have mentioned in this journal's research portion that the project's search algorithm will be simplified. I implemented one of the simplest graphical pathfinding algorithms, Breadth First Search (BFS). This algorithm forms the basis of many more complex pathfinding algorithms, such as Prim's minimum spanning tree and Dijkstra's single source. This algorithm also works well with this project, given that the graph's required properties (such as acquiring a list of adjacent nodes of every element in the board) are easy to achieve. The implementation in this project uses the pseudo code from textbook “Introduction to Algorithms.”[[4]](#footnote-4) By the end of the search, all nodes on the board are given a pointer element, "pi," which indicates the parent node. The sequence of parent nodes will lead the port to the source, and drawing line segments between the parent nodes gives a complete path from the source to the port.

"BFS". Pseudocode for Breadth First Search. Introduction to Algorithms

**Challenges**

           This project had been a lot more complicated than I had imagined in the beginning. Before starting to program, I had yet to think of the conditions that go into circuit designs.

Version 1:

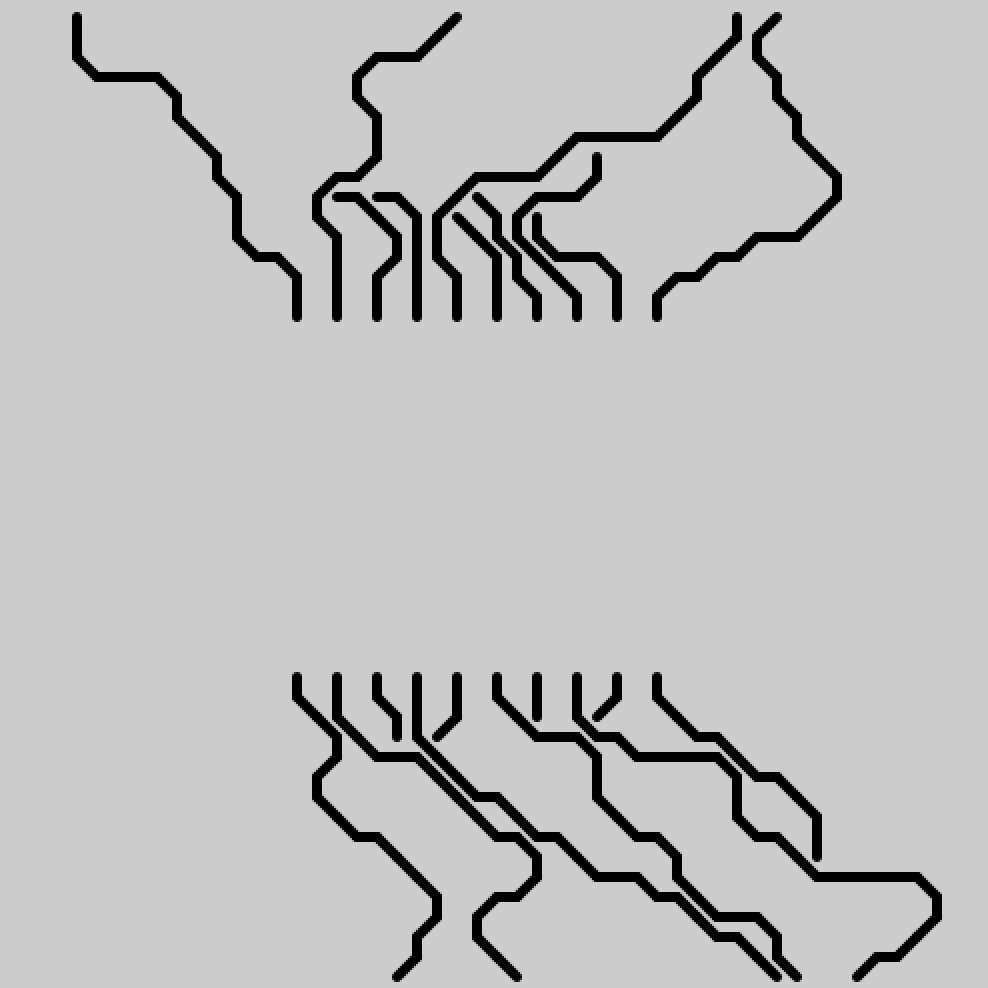
Version 1 of this project did not use the shortest path method. Instead, I chose to determine directions using a random () method. This approach was written without understanding the basics of circuit design (selecting the shortest path) and introduced many difficulties and bugs in the process. Some of the trickiest were writing up cases and conditions dealing with the randomly generated direction, a tedious and frustrating process. It was after one day into this project that I realized that randomly generating paths would not work as there were just too many different cases to work with.

Figure Example of Version 1 Output

Gaps in the Board:

While the shortest path algorithm solved the problem of connecting sources to ports, I needed help finding an excellent method to determine which sources should connect to which port. In the end, I resorted to randomly associating ports to sources. This method introduced the problem of blocking paths that would have existed otherwise, leading to two issues: a messy connection between paths and a board that seems emptier without the appropriate number of paths, i.e., a graph that is not as dense as it could have been.

Performance:

To solve the problem of excess gaps in the board, I decided to create extra "source nodes" in the gaps of the board. The approach was to loop through all nodes not in Union with any path and convert them to individual "sources," testing whether they could connect with the remaining unoccupied ports. This method had undoubtedly filled the board more than before, but it came with a different issue: performance. Since I am testing all non-unionized nodes, I would be running BFS on every one of them. As a result, the program became slower, needing around half a minute to produce the final image.

**Discussion**

There are many aspects of this project that can be improved. I will elaborate on two main issues of this program

Performance:

As explained in “Challenges”, performance issues occurred as the result of an attempt to rid of empty spaces in the image. I can think of two remedies for this problem.

The first being concurrency. One of Java’s most powerful tools is thread manipulation, being able to execute multiple tasks simultaneously. Concurrency was a feature I had worked with the original Java, and since Processing is a Java based language, I would assume it could use the concurrency library just like Java. I did not get a chance to experiment with concurrency in Processing yet so I could not test this hypothesis in this project.

The second being improvement on the algorithm. The BFS algorithm is primitive, with a complexity of O(V + E) where V is the number of nodes and E being the number of possible connection in the graph. The board is 60 by 60 totaling a sum of 3600 nodes with an average of 8 possible connections per node, the calculation load for one run alone is impressively large, and to think that I am running it around 2500 times (one run per node on board, occasionally skipping nodes not feasible as source node, i.e. nodes on path and port nodes). Having a faster algorithm possibly allows for a faster program, but due to limited time for this project, I was not able to find an algorithm more suitable.

Random Selection:

Deviating from the first version of the project, I used a simple pathfinding algorithm to calculate paths from source to ports of other modules. However, as the distribution of source nodes to port nodes is not one-to-one, I faced the problem of assigning ports to sources. This again comes down to the nature of circuit designs, how paths need to be carefully thought through. However, I have not gained enough expertise in this field and thus ended up with randomly assigning sources to ports. This was the most significant cause as to why paths at times seem so unordered and crossing one another at points where there is no reason to. This problem could potentially be solved by longer time dedicated to learning about the patterns of circuit designs, which I have not done during the beginning stages of this project.

**Final Product**

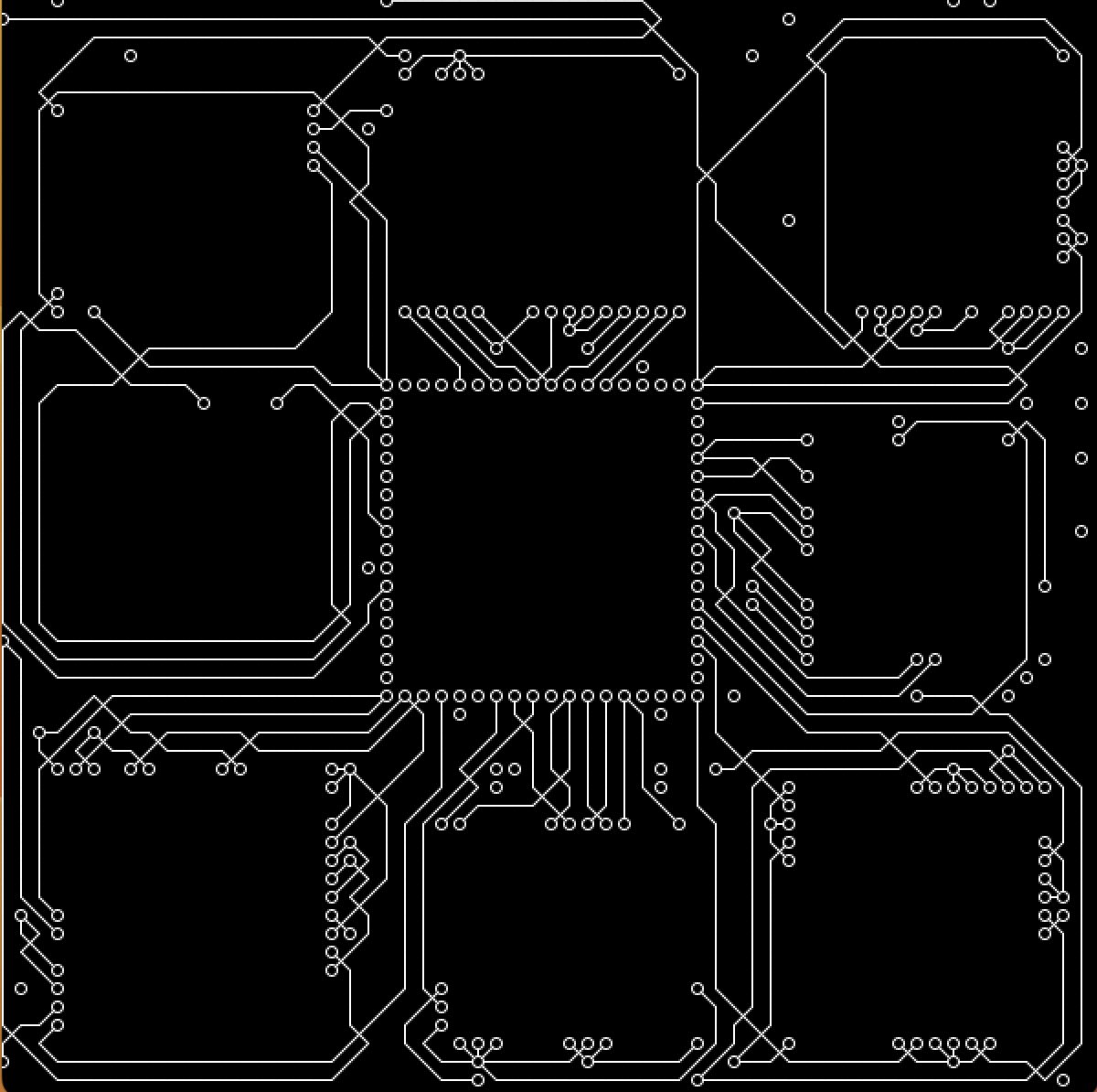
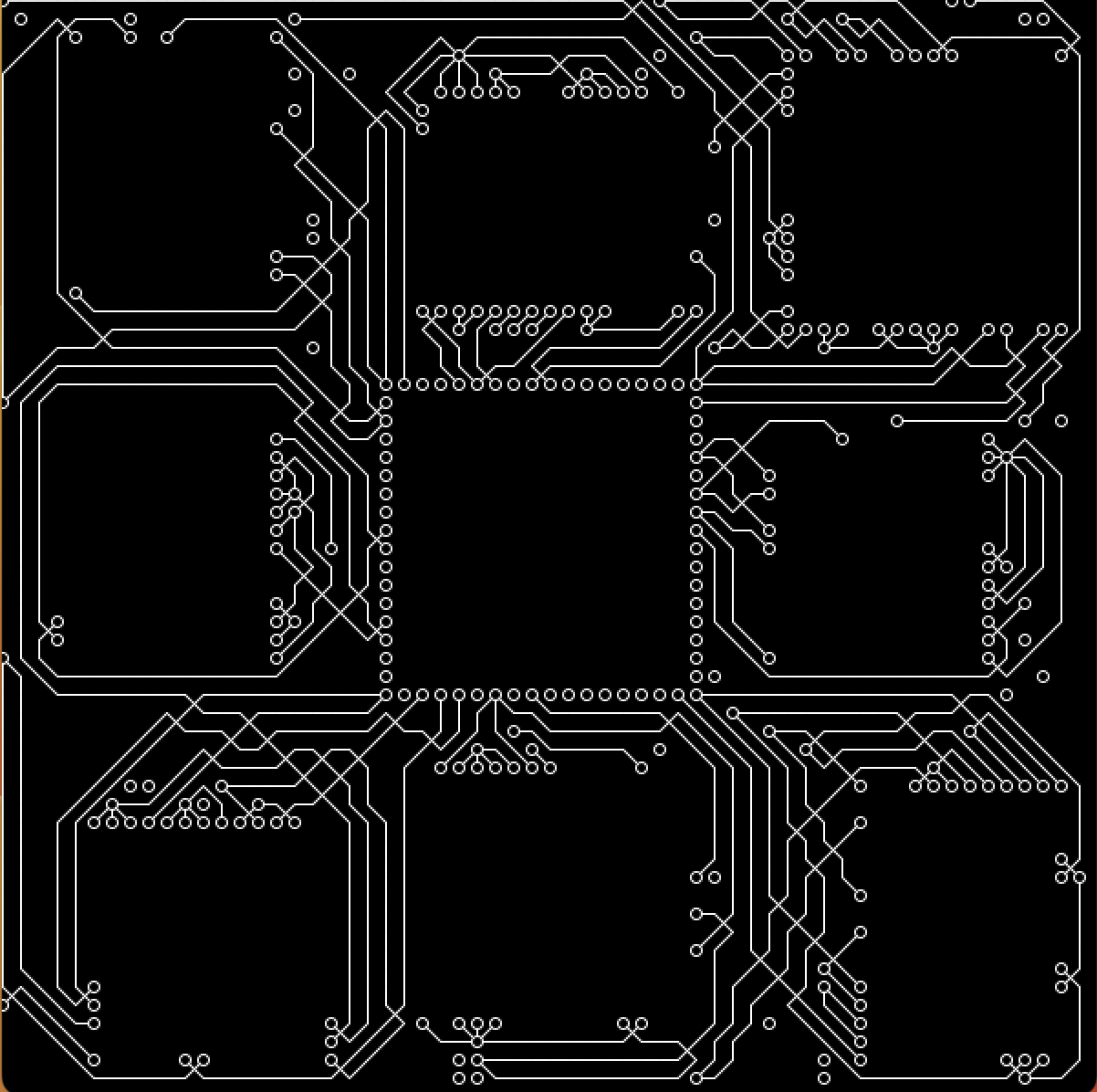
I want to emphasize once again that the program is slow, but that does not mean it is broken code. I have written a progress indicator in the terminal that counts from 0 to 3599. Once the indicator stops at number 3599, that means the image is generated, it should take around 30 seconds.

Figure Examples of Version 2

Bibliography

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to algorithms*. Mit Press.

Danik, I. (n.d.). *Circuit board: Stock vector: Colourbox*. Buy Royalty-free Photos, Images, Videos & Vectors. <https://www.colourbox.com/vector/circuit-board-vector-4549222>

Markoff, J. (2015, September 26). *Smaller, faster, cheaper, over: The Future of Computer Chips*. The New York Times. <https://www.nytimes.com/2015/09/27/technology/smaller-faster-cheaper-over-the-future-of-computer-chips.html>

Patterson, D. A. (2013). *Computer Organization and Design*. Elsevier Science & Technology.

Sedgewick, R., & Wayne, K. (2016). *Computer science: An interdisciplinary approach*. Addison-Wesley.

1. Markoff, *Smaller, faster, cheaper, over: The Future of Computer Chips* 2015. [↑](#footnote-ref-1)
2. Cormen et al., *Introduction to algorithms* 2009: p. 591 [↑](#footnote-ref-2)
3. Sedgewick & Wayne, *Computer science: An interdisciplinary approach* 2016: p. 1014 [↑](#footnote-ref-3)
4. Cormen et al., *Introduction to algorithms* 2009: p. 595 [↑](#footnote-ref-4)