EEE 121: Software Project Documentation

Submitted By:

|  |  |
| --- | --- |
| Kyle Matthew M. Cayanan | 2020-00738 |

I. Milestone 1

How does your final solution work? You can show snippets of your code and define the purpose of each function or line. You can also add a flowchart to visually show the step-by-step procedure.

|  |
| --- |
| **Overall Solution** Full version of each solution to be explained in their own boxes |
| * First, I created a resistor class * This class is meant to store all the attributes of the resistor * Each resistor object is then stored into a dictionary for easy access * The key to this dictionary is the name of the resistor * A modified version of the undirected class is then created   + The weight of each node is now a linked list   + The linked list will contain resistor names   + The resistor names will be used to access the dictionary which contains resistor objects * Each resistor object will then be stored into undirected graph * Upon inputting two resistors, their topology will be checked in this order   + Parallel   + Series   + Neither * In order to check parallel connection   + Check if both resistors have identical nodes   + If yes, then they are parallel * In order to check series connection   + Use a modified version of Dijkstra's Algorithm   + Instead of using number indexes, it default dictionaries with strings as keys   + Instead of checking for weight, it checks if a path is available   + A path is available if and only if there are no branches in between pathways   + Branches mean that a connection is either Neither or Parallel * In order to check special test cases   + Functions were created for the following   + If the Dijkstra's moves into Vdd, then this path is now ended   + If the Dijkstra's moves into GND, then this path is now ended * In order to check if neither   + If both Parallel and Series check fails, that means that it can only be Neither |

|  |
| --- |
| The Resistor Class |
| As seen above the resistor class has four inputs. These four inputs are,   * The resistor name (i.e. R1, R2, etc) * The start node (i.e. Vdd, a, b, c) * The end node (i.e. a, b, c, GND) * The resistance value (i.e. 1000ohms)   The first three inputs are strings. The resistor name is used for identification and for dictionary keys. The nodes in the dictionary are used for the graphs.  Finally, the resistance value is the numeric value that will be used for later milestones. |

|  |
| --- |
| Receiving the Inputs |
| A function for receiving every resistor input is created. This function takes in the number of resistors to be processed as its input. Its output will be the following,   * The dictionary that will be used to access the resistor class objects * The list of resistor names to be used as dictionary keys * The list of nodes to be entered into the graph (outdated for Milestone 2 and 3)     Explanation for V\_list:  In my original implementation, I used normal dictionaries to create the UndirectedGraph. Because of this, I needed to declare every single node in advance even if they are still empty. In later milestones, the implementation of the node is replaced by default dictionaries. This allows me to enter new keys into the dictionary even if they have never been setup yet.    In the code above, I used a for loop in order to process every single resistor object.  First the inputs were mapped into four separate variables.  Then the resistor name and resistor nodes were added into their respective lists.  A resistor class object is created with these input variables, and then deepcopied into the dictionary.  The deep copy was done because everytime the loop repeats, R\_temporary is reset.  Finally after the loop ends, the list of nodes are converted into a set in order to delete duplicates.  These processed variables are now returned. |

|  |
| --- |
| Modified Undirected Graph |
| The undirected graph contains is created using the list of nodes of **all** the resistors. This is because the nodes are in string format instead of integer format. The for loop shows how this is done.  The adj\_list is a dictionary instead of a list. Then the weight of each adj\_list is a linked list that will contain resistor name strings.    In order to add an edge, the resistor name will be appended into the linked list. |

|  |
| --- |
| Modified Djikstra Path |
| In order to be compatible with the software project, the initial vertex is converted into a string instead of an integer. Additionally, all the lists from the original algorithm are converted into dictionaries similar to the undirected graph.    As you can see above, I had to use for loops to enter every single dictionary entry in advance.      The next most important modification in this algorithm is the “valid” boolean found in the self.scan() function. This is the most important variable because it detects whether a path is in series.    The path check function checks two different conditions. Whether there is branching, and whether the current node is Vdd or GND. Each check is explained in more detail below.    When branching occurs between two resistors, that means it is no longer in series and it is no longer a linear path. This is checked by checking the total resistors between a node and its neighbors.  If the linked lists of all neighbors have a total of 2 resistors, then that means it is a linear path.    If the linked lists of all neighbors have 3 or more resistors in total, that means that there is more than one path that can be taken outside of a node. Therefore, it branches.      The next function detects where or not a path has reached its end. In order to prevent false positives, the pathing algorithm is not allowed to use Vdd as a continuation for a path. Once Vdd or GND is reached, it will trigger the return condition to make valid==False.    Without this function, the pathing algorithm will detect the photo above as a valid path.    Finally, the most important change in the self.scan() function is what nodes it scans. Instead of scanning every node that exists in the graph, it only scans the neighboring nodes. This is done via the graph.adj() function. |

|  |
| --- |
| Parallel Checking |
| The solution for parallel checking is simple.  Once the function receives the two resistor names, it can now use the dictionary to call its nodes. The function will then compare whether the two nodes are identical between the resistors and vice versa.  If this happens, parallel=1 is triggered, and then the if-condition will be fulfilled. The if-condition will then return true to the main function which will be shown later. |

|  |
| --- |
| Series Checking |
| Introduction:  Unlike the parallel function, series checking is much more complicated. In order to prevent false positives, several auto fail conditions were put into place.      Autofail condition #1 was created in order to prevent the resistor from starting a pathing algorithm if branching occurs within itself. If branching occurs within itself, then it is automatically impossible that it is in series. For example, NodeA to NodeB is a false positive.    This is the entire function for Autofail condition #1.      If the resistor is cornered by both Vdd and GND, that means it cannot be series with any other resistor. This is because there cannot be a path to any other resistor anymore.  This autofail condition is also important for the series pathing algorithm itself. This is because the rest of the function will completely avoid starting from either Vdd or GND.  The reason why I want to avoid Vdd and GND as the start is because Dijkstra's Algorithm was designed to make all paths neighboring Vdd and GND to be infinite. This can be circumvented temporarily by adding an enable=1 integer inside the function. However, I decided to change my mind and not use this enable variable anymore.    Once all the auto fail conditions have been cleared, each possible node combination is plugged into a helper function which will be explained later.  As you can see above, like I mentioned earlier, the helper functions will only be called if and only if our starting node is neither Vdd or GND.    Finally attached here is the helper function. First the helper function will create a Dijkstra's Algorithm class using the starting node as its input. Remark: the Dijkstra still contains an enable=0 input even though it is no longer needed.  After creating a Dijkstra object, the function then checks again and makes sure that the target node is neither Vdd or GND. Once all of these conditions are cleared, it will call the get\_shortest\_path() function.    The get\_shortest\_path() function has barely any changes. If the path to it is infinite, that means that it is not in series. Otherwise if it will return 1 if it is in series. Since we don’t really care about the actual distance, I simply set the if condition to be “if not none”.    This true statement is returned all the way back to the main function that called it. |

|  |
| --- |
| The Main Function |
| The main function has two main parts.   1. Processing the Inputs 2. Calling the actual Series/Parallel functions   (a) Processing the Inputs     * First the inputs are mapped into their variables * R\_input is called in order to create the resistor class objects * Next the Resistor\_graph is created in order for the series checking needed later * Each resistor name in R\_keys is then used to access the dictionary and add an edge into the resistor\_graph   (b) Calling the actual Series/Parallel functions    For milestone 1, this was done using Q\_count as a way to track the queries. Each query is then entered into the resistor\_check helper function.    This is the helper function. It only uses functions that have all been explained before. If Parallel() is returned “True”, then it return “PARALLEL” as the final string to be printed. If Series() return “True”, then it will print “SERIES”. Finally if none of them are “True”, then it will simply print “NEITHER”.  This is repeated until all queries are complete. |

What was your starting point? Did you use code you made previously?

|  |
| --- |
| Answer |
| My starting point was the algorithms provided in the EEE121 lectures. Afterwards, instead of immediately coding. I started writing down all my ideas in my pseudocode document attached in this [link](https://docs.google.com/document/d/1Qv_wPKrw4tNFeDGtcx1HPu7u02j9AqNdq54CLYnoiU8/edit).  I used this pseudocode in order to keep track of what parts of the EEE121 algorithms I need to modify. It should be noted that I only used this document up to the beginning of Milestone 2. In milestone 2, I noticed that I no longer needed to write down my thoughts. |

From your starting point, what did you need to change to achieve the functionality required?

|  |
| --- |
| Answer |
| (a)  Solution wise, the biggest change I needed to make was to make the algorithms compatible with string inputs as nodes instead of numbers. In order to do this, I used dictionaries for evert node list in the Graph Class and the Dijkstra Class.  (b)  Test case wise, I had to consult with my professor multiple times in order to figure out my mistakes. My biggest mistake was not considering Vdd and GND as endpoints. This is where I got stuck the longest. |

Were there specific test cases you found to be very tricky? How were you able to address them?

|  |
| --- |
| Answer |
| The most difficult test case for me was treating “Vdd” and “GND” as endpoints.  In order to deal with this test case, I had to implement numerous helper functions. For the pathing algorithm, I had to modify all cases of “Vdd” and “GND” to be infinite. For the series pathing, I had to modify it to completely avoid Vdd and GND as the starting point. This sounds simple in concept, but in practice I had to make multiple test cases.  I reviewed my series function and counted 10 different if-cases relating to Vdd and GND alone. |

II. Milestone 2

How does your solution currently work?

|  |
| --- |
| **Overall Solution**  Using milestone 1 as the foundation, I solved this primarily by creating helper functions that checked the topology of each resistor to every other existing resistor. |
| * I improved the efficiency of my code using the guidance of my professor   + Modified all node dictionaries into default dictionaries   + Converted lists to set() whenever allowed * I modified the UndirectedGraph class in order to make it compatible with the helper functions   + I created a self.resistor\_list set() in order to keep track of which resistors have been processed so far   + I added remove\_resistor() in order to remove a resistor from the set() but not the actual edge   + I added remove\_edge() in order to prepare for milestone 3   + I added peek\_edge() in order to check which resistors are **inside** a node   + I added peak\_resistors() in order to check which resistors are **around** a node * I created a Parallel\_Pull() function which takes in a single resistor input and outputs all resistors parallel to it * I created a Series\_Pull() function which takes in a single resistor input, and outputs all resistors series to it * I created print\_helper(). This is a helper function exclusively for the output of milestone 2. This is because if I just print a list as it is, it will contain the quotes(“) in the actual print output |

|  |
| --- |
| Undirected Graph modifications |
| (a) resistor\_list modifications  This data storage was created in order to track which resistors have been processed. If a resistor in the dictionary is not inside this set(), then its processing is skipped.    Function add\_edge() is now modified to accommodate the resistor\_list.  (b) added functions    This function is used to return the resistor(s) inside of the target nodes. For the parallel helper function, this can be used to instantly check all other parallel resistors. For the series helper function, this is used in to check which resistors are neighbors.    This is the actual function that is used to check which resistors are neighbors with the target nodes. It uses the peek\_edge function to check the resistors of all neighbors. Then it returns all these neighboring resistors as a list. |

|  |
| --- |
| Parallel Helper Function |
| The helper function is called Parallel\_Pull(). It takes in a resistor name as its main input, and a dictionary and graph as its supporting data. The destination\_graph variable is a placeholder meant for milestone 3 that was never used.    First, the function uses the dictionary to call the input resistor’s nodes. These nodes are then called into the shallow\_branch() check. Recall that the shallow branch function check if a single set of nodes has more than 1 resistor within it. If this is true, then the enable function is set to True.    Once a set of parallel resistors are detected, the peek\_edge() function is used to return the entire list of resistors.    First, the resistors are removed from the Undirected Graph. This is to improve efficiency and prevent a resistor from being processed more than once.  The resistor is then pushed into a heap as a touple(to be explained in more detail later). Then finally, the total resistance is calculated using the parallel resistor equation.      I will now explain how the minheap works.   1. This solution is implemented using heapq module 2. The resistor name is indexed in order to remove letter “R” from the string. For example, “R12345” will be concatenated by [1:] in order to become “12345”. 3. The touple (“12345”,“R12345”) is pushed into the R\_heap. 4. Next, the touples will be popped one by one. Because this is minheap, the pop will occur in ascending order. 5. Finally, the resistor name (at index=1) in each touple will be appended into the list. |

|  |
| --- |
| Series Helper Function |
| The helper function is called Series\_Pull(). It has the same inputs and outputs as the Parallel\_Pull.    Unlike the parallel helper function, the series function will need to determine which nodes are the start and end. This is why node\_list exists.    These are the variables that will be used by the Series\_Pull function.    The next part of the code is performed using a separate helper function. This helper function is designed to check if the neighboring resistors are series to the starting resistors. This function returns a list because it is possible for the function to return two resistors. It also returns a boolean into “enable” which for the if-conditions.    Here onwards is what the helper function looks like on the inside. It first checks all the neighbor resistors using the peek\_resistors() function. Then it initialized the variables to be returned.    Afterwards, the function first checks if the resistor is inside the resistor\_list. If yes, then that means that it has been processed and should be ignored.  Next it checks if the resistor is inside R\_temp. R\_temp is the set() that the Series\_Pull uses in order to keep track of which resistors have been processed, but have not yet been removed from resistor\_list.  Finally, the function checks whether a resistor is in series using the Series() function from milestone 1. If it succeeds, the helper function will return “True” and the list of resistors.  (**Going back into Series\_Pull**)    Each resistor in the R\_neighbor list is processed. In the worst case scenario, the main resistor is in between these two resistors and both sides need to be processed.  Rc is supposed to represent “current resistor”. As long as the while loop is turned on, it will repetitively check for series connections until “False” is returned. Each success is added into the R\_temp set and R\_heap list.    This is what the inside of the neighbor\_loop() helper function looks like. It is functionally identical to neighbor\_init(), except this time it only returns one resistor. This is because backtracking is not possible. Since the resistor before it has already been processed, it is only possible for it to detect the series resistor after it. In conclusion, it now returns a Bool and a String.  (**Going back into Series\_Pull**)    After all resistors in the chain have been detected, it is now time to actually process them. As shown above, the comment explains what the series\_process() helper function is designed to do. Since the comment already explains it, I will add the series\_process() snippet at the end instead.    Next, the nodes of the series chain are processed. My logic for this solution is that the start node and end node can never occur more than once in a list. This is because all other nodes are the points of connection and therefore exist in 2 resistors at a time.  If a new node is found, it is added. If an old node is found, it is removed. In the end, the only nodes that will remain in the final list is the start and end.    This part of the code is identical to the parallel pull function. It uses a minheap and touple to re-organize every resistor in lexicographical order.    Finally, the function returns the appropriate truth case depending on whether a series detection has succeeded.    Lastly, this is what the series\_process() helper function looked like. |

|  |
| --- |
| Main Helper Function |
| The Get\_Topology() function is designed to receive a single resistor as its main input, and then enter that input into Parallel\_Pull and Series\_Pull. Attached below is the rest of its process.      The most important part of this code is the print\_helper() function. As mentioned in the introduction of Milestone 2, the final print on hackerrank requires a specific format. In order to achieve this format, a helper function was used.    This is the print helper. By using string concatenation and a for loop, each resistor is added into a “list” alongside commas. Finally at the end the resistance value is also added as a string. |

|  |
| --- |
| Main Function |
| The name of the main function is Resistor\_Pull(). It is designed to check the topology of every single resistor in the graph.    First, it performs a for loop in the entire dictionary. Afterwards, it checks if the resistor is inside the resistor\_list graph object. If its not in the list, that means that it has already been processed.  For example, if R1 and R3 are in series, then R3 will be removed from the resistor\_list. R3 will now be skipped.  If it is not skipped, is now entered into the Get\_Topology() helper function. Get\_Topology will return True if its successful, and the final string to be printed. This final string is appended into R\_process.    In the end, this list of strings will be called into the variable “group”. Each string in group will be then be printed. |

Which test cases were not satisfied by your solution? For hidden test cases, you may provide your own test cases which you know your code does not solve correctly.

|  |
| --- |
| Max Resistors Test Case |
| For this test case, I was able to solve it but only after 15 seconds of processing. My code is unable to succeed in less than 10 seconds. |

|  |
| --- |
| Hidden Test Cases |
| I do not know what test cases I could have failed.  Parallel Processing is straightforward so I am confident in it. For series processing, I didn’t modify anything in the original milestone 1 checker. Im not confident because my code is prone to human error, but I honestly didn’t expect problems to arise. |

How does your result for these test cases compare with the expected results?

|  |
| --- |
| Max Resistors Test Case |
| My result succeeded but at 15 seconds |

What would you infer to be the cause of these mistakes and what would you do if given more time?

|  |
| --- |
| Answer |
| (a)  The main reason is because my implementation is not efficient enough. If I had more time, I would try to think of a different solution that doesn’t use Dijkstra. Since I only need to check neighboring resistors maybe I could have implemented a version that simply checks their compatible nodes.  For example, just check if the nodes have branches and if the nodes are Vdd and GND. No need to implement the entire Dijkstra class.  (b)  The reason why I failed in the other test cases must be my carelessness. Milestone 1 already proved that my algorithms for topology checking are correct. I must have made a mistake in my final process/output. |

How much longer do you think it would take to complete this milestone?

|  |
| --- |
| Answer |
| I believe the answer is one week.  I need the first few days to try and implement a version of series check that does not rely on Dijkstra. Then I need the rest of the week for debugging. |

III. Milestone 3

How does your solution currently work?

|  |
| --- |
| **Overall Solution**  This is the milestone which required the least amount of changes in order to become useable. |
| * Modified Series\_Pull and Parallel\_Pull to return nodes instead of only resistors and total weight * Modified Get\_Topology() by disregarding the print\_helper() function * Implemented a while loop to check the number of edges inside the graph * Designed the loop to remove processed resistors and add back the new total resistance within every loop * Loop repeats until the number of edges is only one left * One edge remaining means that only one resistor is left, therefore total resistance has been successfully calculated |

|  |
| --- |
| Modified Helper Functions |
| Snippet from Series\_Pull    Snippet from Parallel\_Pull    Snippet from Get\_Topology  Remarks   * Instead of returning a string, a list is returned * returning R\_nodes is redundant since it is already inside the output list |

|  |
| --- |
| Main Function |
| First the while loop checks if there is more than 1 resistor inside the graph by counting the number of edges. If this is true, the Resistor\_Pull() function is called in order to simplify the graph.    After simplification, each element in the group is processed. The goal is to delete the old resistors, and then insert the total resistance back into the graph. The group element comment explains which part of the list pertains to each index.    If the topology is parallel, the entire edge is removed from a single pair of nodes. The resistance of the main resistor is modified in the dictionary. Then finally, the main resistor is plugged back into the graph.    The if-function for the series topology is almost functionally the same. The main difference is that the nodes of the resistor also change. After removing every resistor from the graph, the changes in the dictionary are applied.  First the dictionary resistance is changed. Then, the dictionary nodes are also changed. Finally the resistor is added back into the graph.    The while loop is repeated until only 1 resistor is left. Everytime a loop is completed, the total resistance is entered into the R\_total variable.    If only 1 resistor is left, then that can only mean that R\_total is the final resistance. |

Which test cases were not satisfied by your solution? For hidden test cases, you may provide your own test cases which you know your code does not solve correctly.

|  |
| --- |
| Max Resistors Test Case |
| I believe that I failed this because my code is still the same as milestone 2. |

|  |
| --- |
| Hidden Test Cases |
| I most likely failed this because of the test cases I have failed to solve in Milestone 2. |

How does your result for these test cases compare with the expected results?

|  |
| --- |
| Max Resistors Test Case |
| My result succeeded but at 16 seconds. |

What would you infer to be the cause of these mistakes and what would you do if given more time?

|  |
| --- |
| Answer |
| I believe that my failures occurred because of the milestone 2 test cases that I was never able to solve. Therefore my answer to this question is that the key to solving milestone 3 is to implement the changes I wanted to make in milestone 2. |

How much longer do you think it would take to complete this milestone?

|  |
| --- |
| Answer |
| I believe the answer is one week.  If milestone 2 is solved within this one week, then milestone 3 will instantly follow. Milestone 3 is just a while loop version of milestone 2. |