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

```
#create resistor class
class R_class:
    def __init__(self, resistor_name, start_node, end_node, resistance):
        #first three inputs can stay as strings
        self.name = resistor_name
        self.node1 = start_node
        self.node2 = end_node

        #resistor in Ohms needs to be converted to float
        self.weight = float(resistance)
```

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)

```
def R_input(n_resistors):
    #declare initial dictionary
    R_dictionary={}

    #declare initial key list
    R_keylist=[]

    #declare initial vertex list
    V_list=[]

    #declare loop counter
    r_count = 0
```

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.

```
    #repeat loop for every input of R
    while r_count < n_resistors:
        #iterate r_count
        r_count+=1

        #input split the resistor values
        #take the input as string
        name, N1, N2, W = map(str, input().split())

        #enter the name(key) into a list
        R_keylist+=[name]

        #enter the nodes into the vertex list
        V_list+=[N1,N2]

        #create a resistor object
        R_temporary = R_class(name,N1,N2,W)


        #enter the class into a dictionary using deepcopy
        R_dictionary[name] = copy.deepcopy(R_temporary)


    #convert the list of nodes into a set, then back into a list
    V_return = list(set(V_list))


    return R_dictionary, R_keylist, V_return
```

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.

```
return R_dictionary, R_keylist, V_return
#final output:
#R_dictionary = Dictionary of R_classes
#R_keylist = List of Resistor Names(Keys)
#V_return = List of Nodes
```

Modified Undirected Graph

```python
class GraphUndirected:

    #modify the graph to use dictionaries instead of only lists
    def __init__(self, vertex_list):
        #name change
        self.vertex_list = vertex_list

        #no change
        self.num_edges = 0

        #changed to a dictionary
        #to minimize user errors: the name of adj_list was not changed
        self.adj_list = {}

        #add the nodes into the main "adj_list" dictionary
        for vertex in vertex_list:
            #each vertex will be added as a dictionary
            self.adj_list[vertex] = {}

            for neighbor in vertex_list:
                #each neighbor will point to an empty list
                self.adj_list[vertex][neighbor] = []
```

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.

```python
#change function to append()
def add_edge(self, v: str, w: str, weight: str):
    #explanation:
    #the weight of the nodes must be a linked list in order to accomodate parallel connections
    #the standard list in python must either use "+[value]" or .append(value)
    self.adj_list[v][w].append(weight)
    self.adj_list[w][v].append(weight)
    self.num_edges += 1
```

In order to add an edge, the resistor name will be appended into the linked list.

Modified Djikstra Path

```python
####################################
#Modified Dejikstra Path
####################################

#modify for exclusive resistor use
class Dijkstra:

    #change initial vertex to voltage, Vdd
    def __init__(self, graph: GraphUndirected, init_v="Vdd", enable=1):
        #initial variables
        self.graph = graph #undirected resistor graph
        self.source = init_v #this is now a string
```

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.

```
#modify dist_to into a dictionary
self.dist_to = {}
self.edge_to = {}
self.is_marked = {}
for i in graph.vertex_list:
    self.dist_to[i] = float('inf')

#initialize edges as NONE
    self.edge_to[i] = None

#initialize markings as FALSE
for i in graph.vertex_list:
    self.is_marked[i] = False
```

As you can see above, I had to use for loops to enter every single dictionary entry in advance.

```
    #modify to check for branching paths
    def scan(self, graph: GraphUndirected, pq: MinHeap, point: str, enable=1):
        #mark check
        self.is_marked[point] = True
```

```
#perform validity check
#def path_check(v,w,graph):
valid = path_check(point, neighbor, graph, enable) #recall: path_check verifies Vdd, GND, and branching paths

#if valid==True, that means there are no branches, VDD, or GND
#if valid==False, that means an invalid path was found
```

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.

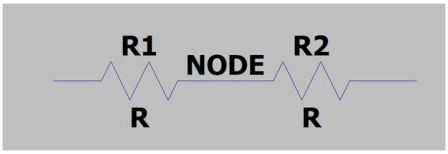
```python
def path_check(v,w,graph, enable=1):
    branch = branch_check(v, w, graph) #check if there are more than three paths into the node
    node = node_check(v, w, graph) #check if the current node is Vdd or GND

    #if either of the invalid paths are triggered, return False
    if(branch):
        return False

    if(node and enable==1):
        return False

    #else return True
    return True
```

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.

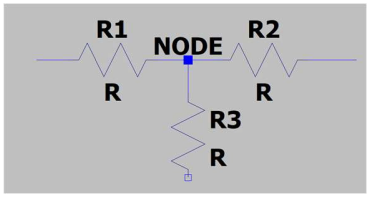
```python
#branches are invalid path for series
def branch_check(v,w,graph):
        #skip self check
        if v==w:
            pass

        #check length of linked list
        #redefine weight_check: a branch exists when a node has at least 3 connections to ALL its neighbors

        weight_check = 0

        #get the total number of resistors in all neighbors
        for neighbor in graph.adj_list[v]:
            weight_check += len(graph.adj_list[v][neighbor])

        #if resistors are greater than 2, then there is a branch in the path
        if weight_check > 2:
            return True

        #else return False
        return False
```

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.

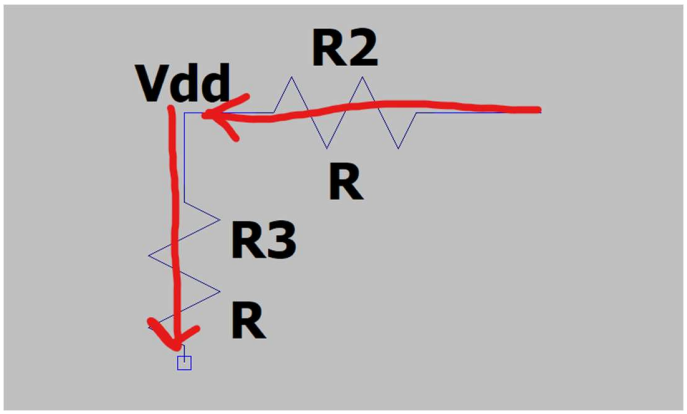


If the linked lists of all neighbors have 3 or more resistors in total, that means that there is more than one path that can be taken outside of a node. Therefore, it branches.



```
def node_check(v,w,graph):
        #the node to itself is an "invalid path"
        if v==w:
            return True

        #if Vdd, then next node is infinity
        if(v=="Vdd"):
            return True

        #if GND, then next node is infinity
        if(v=="GND"):
            return True

        #else return False
        return False
```

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.

```
#modify if case to check for valid paths instead of caring about smaller weight
if(valid):
    ######################################################
    #Major Change: add adjacent nodes of neighbor to pq
    ######################################################
    for v in graph.adj(neighbor):

        #make sure that you dont re-add already visited locations
        if self.is_marked[v]==False:
            #add the key string as infinite float
            pq.add(v, float('inf'))


    #before: in the past, Djikstra would visit every node regardless if it is impossible to visit
    #now: djikstra will only visit its neighbors, and the neighbors of its neighbors

    #conslusion: djikstra will now traverse VALID paths only

    #let distance = 1 if path is valid
    #no need to add
    self.dist_to[neighbor] = 1

    #no change
    self.edge_to[neighbor] = point

    #same as dist_to[neighbor]
    #just let it be 1
    pq.add(neighbor, 1)
```

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

```
###################################
# Parallel Check
###################################

def Parallel(start, end, graph, dictionary):
    #declare parallel variable
    parallel = 0

    #call the resistor objects
    R1 = dictionary[start]
    R2 = dictionary[end]

    #declare the nodes to be compared
    node11 = R1.node1
    node21 = R2.node1
    node12 = R1.node2
    node22 = R2.node2

    #first check: starting nodes are equal
    if(node11==node21 and node12==node22):
        parallel = 1

    #second check: opposite nodes are equal
    if(node12==node21 and node11==node22):
        parallel = 1

    #final check, return True if either checks are satisfied
    if(parallel==1):
        return True
    #else return False
    return False
```

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.

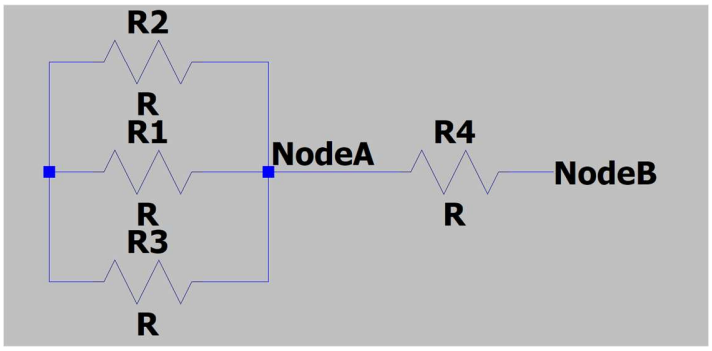
If this happens, parallel=1 is triggered, and then the if-condition will be fulfilled. The if-condition will then return true to the main function which will be shown later.

Series Checking

```
: ###################################
  # Series Check
  ###################################

  def Series(start, end, graph, dictionary):
      #call the resistor objects
      R1 = dictionary[start]
      R2 = dictionary[end]

      node11 = R1.node1
      node21 = R2.node1
      node12 = R1.node2
      node22 = R2.node2
```

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 check#1: branching
#if there is a branch between a single resistor's two nodes, then it is automatically neither
if(shallowbranch_check(node11,node12,graph) or shallowbranch_check(node21,node22,graph)):
    return False
```



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 meant to check if there is a branch between the node itself
# "why does this matter?"
#because: it is possible that only one side of the resistor has branches while the other side is still series
def shallowbranch_check(v,w,graph):
        #skip self check
        if v==w:
            pass

        #check length of linked list
        weight_check = len(graph.adj_list[v][w])

        #if list is greater than 1, then there is a branch within the resistor nodes
        if weight_check > 1:
            return True

        #else return False
        return False
```

This is the entire function for Autofail condition #1.

```
    #autofail check#2: cornered by Vdd and GND
    #if a single resistor is cornered by these two circuit points, it is automatically neither
    if(VG_check(node11,node12) or VG_check(node12,node22)):
        return False
```

```
###################################
# Autofail Condition: Vdd and GND
###################################
def VG_check(n1,n2):
    if(n1=="Vdd" and n2=="GND"):
        return True
    if(n1=="GND" and n2=="Vdd"):
        return True

    #else
    return False
```

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.

```
    #test check for node11
    if(node11!="Vdd" and node11!="GND"):
        #path finder route 1: node11 to node21, then validity of node21 to node22
        if(Series_Pathing(node11,node21,node22,graph)):
            #print("route1")
            return True
        #path finder route 2: node11 to node22, then validity of node22 to node21
        if(Series_Pathing(node11,node22,node21,graph)):
            #print("route2")
            return True

    #test check for node12
    if(node12!="Vdd" and node12!="GND"):
        #path finder route 3: node12 to node21, then validity of node21 to node22
        if(Series_Pathing(node12,node21,node22,graph)):
            #print("route3")
            return True
        #path finder route 4: node12 to node22, then validity of node22 to node21
        if(Series_Pathing(node12,node22,node21,graph)):
            #print("route4")
            return True

    #else return False
    return False
```

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.

```
###################################
# Series Pathing
###################################
def Series_Pathing(n1,n2,n3,graph):

    #create modified Djikstra Object using R1 as the init_v
    Pathing = Dijkstra(graph, n1, 0) #recall def __init__(self, graph: GraphUndirected, init_v="Vdd"):

    #only allow pathing if the destination is not Vdd or GND
    #declare series_check1 and series_check2
    series_check1=None
    series_check2=None


    if(n2!="Vdd" and n2!="GND"):
        #find a path from n1 to n2
        series_check1, series_path1 = Pathing.get_shortest_path(n2)
        branch1 = branch_check(n2,n3,graph) #then check for branching

    if(n3!="Vdd" and n3!="GND"):
        series_check2, series_path2 = Pathing.get_shortest_path(n3)
        branch2 = branch_check(n3,n2,graph) #then check for branching

    #path is true if and only if, there is a path and there is no branching between the pair
    if(series_check1!=None and branch1==False):
        return True
    if(series_check2!=None and branch2==False):
        return True

    #else return False
    return False
```

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.

```
def get_shortest_path(self, dest):
    #starting vertex (ex) "Vdd"
    start = self.source

    #change: transferred the infinite return case to the beginning in order to reduce runtime

    #if there is no path, return None
    #print(self.dist_to[dest])
    if(self.dist_to[dest]==float('inf')):#if None is returned, then it is not in Series
        return None, None


    #target destination vertex (ex) "GND"
    end = dest

    #declare empty path list
    path = []

    #no changes needed
    while end is not None:  # Check if end is not None
        path.append(end)
        end = self.edge_to[end]

    #no changes needed
    path.append(start)
    path.reverse()



    return self.dist_to[dest], path
```

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".

```
if(series_check1!=None and branch1==False):
    return True
if(series_check2!=None and branch2==False):
    return True
```

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.

(a) Processing the Inputs

(b) Calling the actual Series/Parallel functions

(a) Processing the Inputs

```
#resistor and queries
R, Q= map(int, input().split())

#R = number of resistors
#Q = number of queries


#call the resistor processing function
#return a dictionary, keylist, and vertex list
R_dict, R_keys, R_nodes = R_input(R)

#create the resistor graph
Resistor_graph = GraphUndirected(R_nodes)

#add the resistors as edges to the graph
for resistor in R_keys:
    Resistor_graph.add_edge(R_dict[resistor].node1,R_dict[resistor].node2,R_dict[resistor].name)
```

- 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

```
#perform queries
Q_count = 0
while Q_count<Q:
    #increment
    Q_count+=1

    #map the string keys
    Key1, Key2 = map(str, input().split())

    #call the checker function
    #recall: Resistor_check(start, end, graph, dictionary):
    Path = Resistor_check(Key1, Key2, Resistor_graph, R_dict)

    print(Path)
```

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.

```
##################################
# Resistor Check
##################################

def Resistor_check(start, end, graph, dictionary):

    #first check parallel
    #if parallel is returned True, then return the parallel statement
    if(Parallel(start, end, graph, dictionary)):
        return("PARALLEL")

    #if series is returned True, then return series
    if(Series(start, end, graph, dictionary)):
        return("SERIES")

    #if none of the checks are true, then it will be neither
    return("NEITHER")
```

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. 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

```
#new init variable
self.resistor_list=set()
```

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.

```
#change function to append()
def add_edge(self, v: str, w: str, weight: str):
    #explanation:
    #the weight of the nodes must be a linked list in order to accomodate parallel connections
    #the standard list in python must either use "+[value]" or .append(value)
    self.adj_list[v][w].append(weight)
    self.adj_list[w][v].append(weight)
    self.num_edges += 1

    #change function to append()

    #new action: add to resistor list
    self.resistor_list.add(weight)
```

Function add_edge() is now modified to accommodate the resistor_list.

(b) added functions

```
###################################
#New Function: peek edge
###################################
def peek_edge(self, v: str, w: str):
    #take a peek of the resistor(s) INSIDE two nodes
    R_out = self.adj_list[v][w]

    return R_out
```

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.

```
################################
#New Function: peek resistors
################################
def peek_resistors(self, a, b):
    #take a peek at all the resistors AROUND two nodes
    resistors = []

    #get adjacent nodes
    neighbors_a = self.adj(a)
    neighbors_b = self.adj(b)

    #process both sides
    for w in neighbors_a:
        if(w==b):
            pass
        #else
        temp = self.peek_edge(a,w)
        resistors+=temp
    for w in neighbors_b:
        if(w==a):
            pass
        #else
        temp = self.peek_edge(b,w)
        resistors+=temp


    #return the list of resistors
    return resistors
```

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.

```
def Parallel_Pull(resistor_name, target_graph, dictionary, destination_graph=None):
    #call the resistor object
    R0 = dictionary[resistor_name]

    #declare nodes
    nodeA=R0.node1
    nodeB=R0.node2

    #perform a shallow branch check
    #if it is returned TRUE, then the resistor is parallel with someone
    enable = shallowbranch_check(nodeA, nodeB, target_graph)

    #declare variables to be returned
    R_list = []
    R_eq = float(0)
    R_heap = []
```

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.

```
#if enabled, perform solving
if(enable):
    #pop the parallel resistors
    #Resistors = target_graph.remove_edge(nodeA,nodeB)

    #change: get the resistors instead of popping them
    Resistors = target_graph.peek_edge(nodeA,nodeB)

    #change: you're not allowed to pop the resistors to prevent false positives on NEITHERS
    #instead: just remove them from the resistor list
```

Once a set of parallel resistors are detected, the peek_edge() function is used to return the entire list of resistors.

```
#process each resistor
for r in Resistors:
    target_graph.remove_resistor(r) #def remove_resistor(self, Resistor):

    #push every resistor into a minheap
    heapq.heappush(R_heap,(int(r[1:]),r))

    #continue with solving for total resistance
    resistor = dictionary[r]
    ohms = dictionary[r].weight
    R_eq+= 1/ohms

#finalize Req
R_eq = (R_eq)**(-1)
```

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.

```
    #push every resistor into a minheap
    heapq.heappush(R_heap,(int(r[1:]),r))
```

```
#finalize the output list
popsize = len(R_heap) #length of the minheap list
R_pop = None #variable that will hold onto popped touple

for i in range(popsize):
    #pop the touple
    R_pop = heapq.heappop(R_heap)

    #at the resistor name from index=1 into the final R_list
    R_list.append(R_pop[1])
```

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.

```
#finally, return all the relevant values
return enable, R_list, R_eq
```

Series Helper Function

The helper function is called Series_Pull(). It has the same inputs and outputs as the Parallel_Pull.

```
def Series_Pull(resistor_name, target_graph, dictionary, destination_graph=None):
    #purpose of dict_enable: only modify the dictionary during the top level recursion

    #call the resistor object
    R0 = dictionary[resistor_name]

    #declare nodes
    nodeA=R0.node1
    nodeB=R0.node2

    #declare nodes list
    node_list = [nodeA, nodeB]
```

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.

```
#declare variables to be returned
R_heap = [] #list that will be used to order resistors in ascending order
R_temp = set([R0.name])#set that will contain all processed resistors so far
R_list = []#final output list
R_eq = float(0)#total resistance so far
enable = False
check = False
```

These are the variables that will be used by the Series_Pull function.

```
###############################################################################
# Change: only perform series checking on neighbors
###############################################################################

#perform initial series check
#recall: def neighbor_init(Rsource:str, R_temp:set, target_graph, dictionary):
#recall: return check(Bool), r_out(List)

#check if the resistor has series neighbors
enable, R_neighbor = neighbor_init(R0.name, R_temp, target_graph, dictionary)
```

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.

```
################################
# Helper Functions for Series Pull
################################

#check each neighbor of the MAIN resistor
#adds series resistors to the R_tempt set()
def neighbor_init(Rsource:str, R_temp:set, target_graph, dictionary):
    #get neighboring resistors
    #recall: def peek_resistors(self, a, b):
    neighbors = target_graph.peek_resistors(dictionary[Rsource].node1, dictionary[Rsource].node2)

    #declare empty variables
    r_out = []
    check = False
```

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.

```
for R1 in neighbors:
    #skip if not in resistor list
    if(R1 not in target_graph.resistor_list):
        #because already processed
        continue

    #skip if already in R_temp
    if(R1 in R_temp):
        #because already processed
        continue

    if(Series(Rsource, R1, target_graph, dictionary)):#recall: def Series(start, end, graph, dictionary):
        #set to True permanently
        check = True

        #add to set
        R_temp.add(R1)

        #add to output return
        r_out.append(R1)

    #else nothing happens

return check, r_out
```

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**)

```
#if enable is true, then perform series_pull on its neighbors
#now process the neighbors
if(enable):

    #process the one or two resistors in series with the main resistor
    for Rn in R_neighbor:
        #turn ON the while loop
        check = True

        #let the current resistor (Rc) be the initial neighbor(Rn)
        Rc = Rn

        while(check):

            #check if the current resistor has any series neighbors
            check, Rc = neighbor_loop(Rc, R_temp, target_graph, dictionary)

            #if there is at least one series neighbor, check==True, Rc= {Series Neighbor}
            #the while loop will repeat and process {Series Neighbor}

            #if there are no series neighbors, check==False and Rc==None
            #the while loop will break and move on to the second resistor in R_neighbor
```

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.

```
#modified version of neighbor_init()
#guaranteed to only return one series resistor at a time
#proof:there can only be two series neighbors at one time;  function ignores the already processed resistor in the line
def neighbor_loop(Rsource:str, R_temp:set, target_graph, dictionary):
    #get neighboring resistors
    #recall: def peek_resistors(self, a, b):
    neighbors = target_graph.peek_resistors(dictionary[Rsource].node1, dictionary[Rsource].node2)

    #declare empty variables
    r_out = None
    check = False

    for R1 in neighbors:
        #skip if not in resistor list
        if(R1 not in target_graph.resistor_list):
            #because already processed
            continue

        #skip if already in R_temp
        if(R1 in R_temp):
            #because already processed
            continue

        if(Series(Rsource, R1, target_graph, dictionary)):#recall: def Series(start, end, graph, dictionary):
            #set to True permanently
            check = True

            #add to set
            R_temp.add(R1)

            #add to output return
            r_out = R1

        #else nothing happens
    return check, r_out
```

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**)

```
#now process all the series resistor names
if(enable):

    ##########################################################################################################
    #process every series resistor found
    #recall: every resistor in R_temp is already confirmed to be series
    for R1 in R_temp:
        #recall: def series_process(R1, node_list, R_heap, R_eq, target_graph, dictionary):
        R_eq = series_process(R1, node_list, R_heap, R_eq, target_graph, dictionary)

        #purpose:
        #(a) add R1 nodes to node_list
        #(b) add R1 name to R_heap
        #(c) add R1 resistance to R_eq
        #(d) remove R1 name from graph resistor_list
```

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: process the nodes
final_nodes=[]

#for loop to look for the start node and end node of the chain of series resistors
for node in node_list:
    #add the node to the final list
    if(node not in final_nodes):
        final_nodes.append(node)

    #if the node is found again, that means it is not in the edge of the series
    #delete it
    elif(node in final_nodes):
        final_nodes.remove(node)
```

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.

```
#########################################################################################################
#next: organize the resistors in increasing order

#OPTIMIZE 2.2: finalized the sorting for R_list

#add the sorted resistor names to the final list
#recall: {tuple at index 1} = Resistor Name

popsize = len(R_heap) #length of the minheap list
R_pop = None #variable that will hold onto popped touple

for i in range(popsize):
    #pop the touple
    R_pop = heapq.heappop(R_heap)

    #at the resistor name from index=1 into the final R_list
    R_list.append(R_pop[1])
```

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.

```
    return True, R_list, R_eq


#finally, return all the relevant values
return False, R_list, R_eq
```

Finally, the function returns the appropriate truth case depending on whether a series detection has succeeded.

```
#add the series resistor to the total list
def series_process(R1, node_list, R_heap, R_eq, target_graph, dictionary):

    #add R1 nodes to nodelist
    #node_list+=[dictionary[R1].node1, dictionary[R1].node2]
    node_list.append(dictionary[R1].node1)
    node_list.append(dictionary[R1].node2)

    #OPTIMIZE 2.1: changed R_list into R_heap in order to perform sorting on the phonetics via string concatenatio
    # string="R123" ; string[1:] = "123"

    #add R1 name to the minheap
    heapq.heappush(R_heap,(int(R1[1:]),R1))

    #add R1 resistance to final output
    R_eq += dictionary[R1].weight

    #remove R1 name from resistor list
    target_graph.remove_resistor(R1) #def remove_resistor(self, Resistor):

    return R_eq
```

Lastly, this is what the series_process() helper function looked like.

Main Helper Function

```
##########################################
# Topology Helper Function
##########################################
def Get_Topology(resistor, graph, dictionary, destination_graph=None):
    #declare variables
    allow = False
    R_list = []
    R_eq = float(0)
```

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.

```
################################
#call series processing
################################

#then return if true
allow, R_list, R_eq = Series_Pull(resistor, graph, dictionary)
if(allow):
    #identical to parallel if-case
    R_eq = int(R_eq)

    R_first = int(R_list[0][1:])

    output_string = print_helper(R_list, R_eq)

    return True, output_string, R_first
```

```
################################
#call series processing
################################

#then return if true
allow, R_list, R_eq = Series_Pull(resistor, graph, dictionary)
if(allow):
    #identical to parallel if-case
    R_eq = int(R_eq)

    R_first = int(R_list[0][1:])

    output_string = print_helper(R_list, R_eq)

    return True, output_string, R_first

################################
#neither processing
################################

return allow, None, None
#where allow==False
```

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.

```
########################################
# String Output Helper Function
########################################

def print_helper(rlist, requivalent):
    test = rlist
    out = "["
    oot = str(requivalent)
    for i in range(len(test)-1):
        uut = test[i]
        out+= uut
        out+=", "

    out+=test[-1]
    out+="]"
    out+=" "+oot
    return(out)
```

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.

```
########################################
# Final Function
########################################
def Resistor_Pull(graph, dictionary, destination_graph=None):
    #create main output list
    R_process = []

    #DEBUG
    count=0

    #check every resistor in the dictionary
    for R in dictionary:
        #debug string
        #print(f'count{count} start')

        #if R exists in the resistor_list: then it has not been popped yet
        #why popping: to prevent duplicates from being processed
        if(R in graph.resistor_list):

            #process R
            allow, string, integer = Get_Topology(R, graph, dictionary)

            #if a topology is detected, add to list
            if(allow):
                #R_process+=[string]
                R_process.append(string)
```

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.

```
#call the grouping function
#recall: def Resistor_Pull(graph, dictionary, destination_graph==None):

group = Resistor_Pull(Resistor_graph, R_dict)

#observe: I did not put a test case for if "group" is empty
#reason: It is physically impossible for all resistors to be "Neither" unless Wye Delta

for i in group:
    print(i)
```

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

```
    return True, R_list, R_eq, final_nodes

final_nodes=[]
#finally, return all the relevant values
return False, R_list, R_eq, final_nodes
```

Snippet from Series_Pull

```
#finally, return all the relevant values
return enable, R_list, R_eq, R_nodes
```

Snippet from Parallel_Pull

```
#convert the data into the required string format: [R1, R2, R3] 1000
output_list = [R_list, R_eq, 'Parallel', R_nodes]
return True, output_list, R_nodes
```

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

```
while(Resistor_graph.num_edges!=1):
    group = Resistor_Pull(Resistor_graph, R_dict)
    #print(group)

    #observe: I did not put a test case for if "group" is empty
    #reason: It is physically impossible for all resistors to be "Neither" unless Wye Delta
```

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.

```
for i in group:
    #recall group element format: [ [R1, R2] , 2000 ,'Series' , [nodea,nodeb]]

    #get main resistor
    MainName=i[0][0]
    MainResistor=R_dict[MainName]

    #get group elements
    TotalResistors=i[0]
    TotalResistance=i[1]
    topology=i[2]
    nodes=i[3]
```

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(topology=='Parallel'):
    #remove resistors from graph
    #recall: def remove_edge(self, v: str, w: str):
    Resistor_graph.remove_edge(nodes[0],nodes[1])

    #modify dictionary weight
    MainResistor.weight=TotalResistance

    #add the main resistor back into the graph
    Resistor_graph.add_edge(MainResistor.node1,MainResistor.node2,MainResistor.name)
```

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.

```
if(topology=='Series'):
    #remove all resistors from graph
    for reseries in TotalResistors:
        #recall: def remove_edge(self, v: str, w: str):
        Resistor_graph.remove_edge(R_dict[reseries].node1,R_dict[reseries].node2)

    #modify dictionary weight
    MainResistor.weight=TotalResistance

    #modify dictionary nodes
    MainResistor.node1=nodes[0]
    MainResistor.node2=nodes[1]

    #add the main resistor back into the graph
    Resistor_graph.add_edge(MainResistor.node1,MainResistor.node2,MainResistor.name)
```

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.

```
R_total=TotalResistance
#repeat until only 1 resistor left
```

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.

```
print(int(R_total))
```

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.