MS 1

Wednesday, 28 June 2023 4:46 pm

STORING THE INPUT:

The program starts by parsing the input into a nested dictionary called resList. The resistor name will be the key of the outer dictionary and V1, V2, and resistance are the keys of the inner dictionary. This allows an easy access to resistor information which is necessary in the execution of the succeeding codes.

```
#takes in the first line of user input
N, Q = map(int, input().split())

#transforms the input into a list of resistors called 'resList' by calling def dictResistors
resList = {}
for i in range(N):
    r_info = input().split()
    dictResistors(resList, r_info)
```

```
#function that sets-up a nested dictionary where the resistor name is the key for the outer dictionary
#and the information on nodes and resistance are key-value pairs as shown below
def dictResistors(resList, r_info):
    key = r_info[0]
    resList[key] = {
        "V1" : r_info[1],
        "V2" : r_info[2],
        "resistance" : int(r_info[3])
    }
```

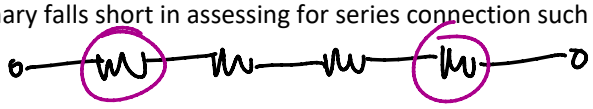
CHECKING FOR PARALLEL CONNECTION:

```
#evaluates the two resistors in the query
for j in range(Q):
    res1, res2 = input().split()
    vertices_res1 = [resList[res1]["V1"], resList[res1]["V2"]]
    vertices_res2 = [resList[res2]["V1"], resList[res2]["V2"]]
    if(set(vertices_res1) == set(vertices_res2)):
        print("PARALLEL")
```

Parallel connections are easy to evaluate when using the resistor dictionary above. If the vertices V1 and V2 of the two resistors match, then they are parallel. If the query fails the parallel connection test, we'll then proceed to the series test. Note that two resistors cannot be both series and parallel at the same time so an if-else conditional statement can be used.

PROBLEM ENCOUNTERED:

The resistor dictionary falls short in assessing for series connection such as in cases below:



Thus, a graph is implemented.

CHECKING FOR SERIES CONNECTION:

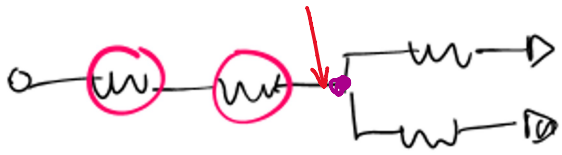
For evaluating the series connection, the following logic was used:

It starts with the assumption that the two resistors are in series (series = 1). The program will then run through various conditions that if satisfied, will negate this initial assumption (series = 0).

- If a resistor is in between Vdd and GND, then it is impossible for it to have any series connection. It also falls accordingly that we check for parallel connections first; if the two resistors are both connected to Vdd and GND, this will be detected by the parallel connection test.

```
if vertices_res1 == ['Vdd', 'GND'] or vertices_res2 == ['Vdd', 'GND']:
    series = 0
```

- If the program bypasses the above condition, then the shortest path between the two resistors is determined. Since BFS functions by searching for vertices, V2 of resistor1 will be used as the starting vertex and V1 of resistor2 will be used as ending vertex. Why V1 of resistor2 and not V2? The two resistors below are in series. If we include the lower vertex of the second resistor (V2), then the connection will be marked as “NEITHER” due to the parallel connection in the lower vertex.

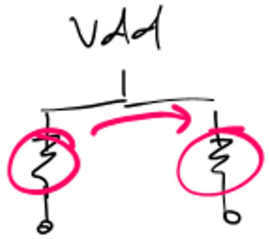


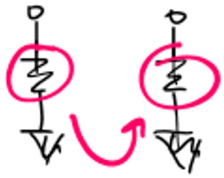
The logic as to why we use V2 of resistor 1 is best discussed later.

- The succeeding conditions that negate that two resistors are in series are best explained with illustrations as seen below. The Vdd and GND nodes do not function similar to other nodes so it is important to take note of them if the shortest path from resistor1 to resistor2 passes by either Vdd or GND.

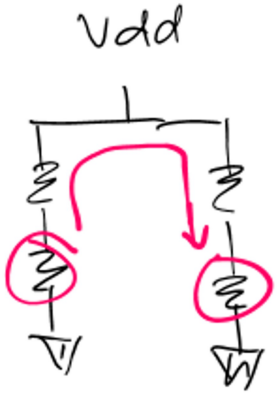
```
else:
    path = shortest_path(resGraph, vertices_res1[1],  vertices_res2[0])
    for i in path:
        #case1
        if len(path)==1 and (i=="Vdd" or i=="GND"):
            series = 0
            break
        #case2
        if i != path[0] and (i == "Vdd" or i == "GND"):
            series = 0
            break
        else:
            #case3
            if len(resGraph[i].keys()) > 2:
                series = 0
                break
            #case4
            for neighbors in resGraph[i]:
                if len(resGraph[i][neighbors]) >= 2:
                    series = 0
                    break
```

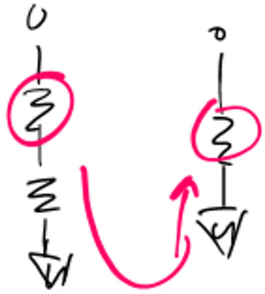
CASE 1:



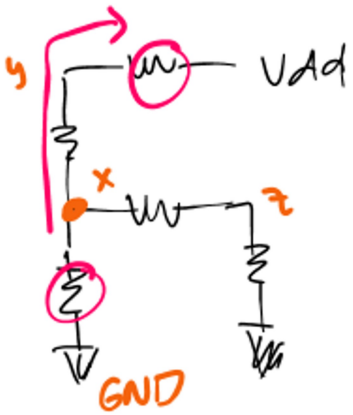


CASE 2:



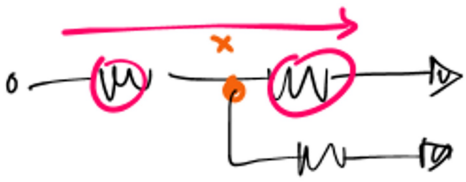


CASE 3:



Vertex x has more than two neighbors: {GND, y, z}

CASE 4:



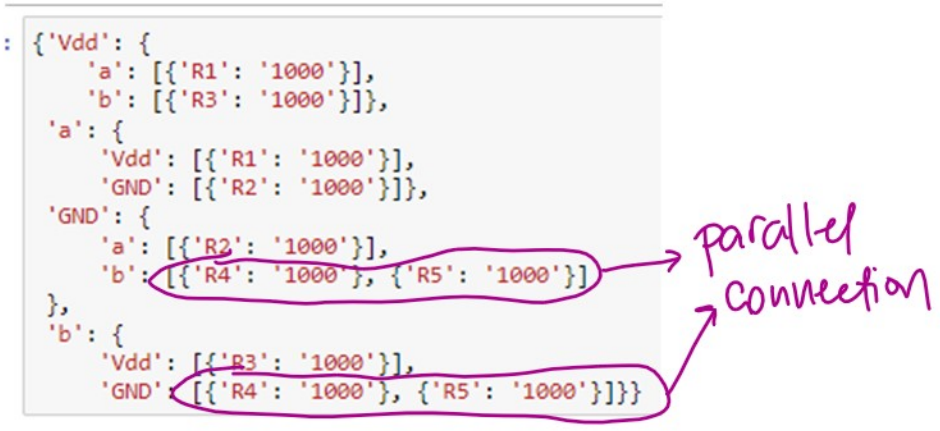
Vertex x has parallel connection

EXPOUNDING ON CASE 4

To denote a parallel connection in the adjacency list, the parallel resistor will be appended as a new dictionary inside the list of the inner dictionary. Parallel connections are not stored as separate vertices, so they will be bypassed by the Case 3 condition.

Thus for case 4, it is necessary to inspect how many elements are inside the inner dictionary to check for parallel connections.

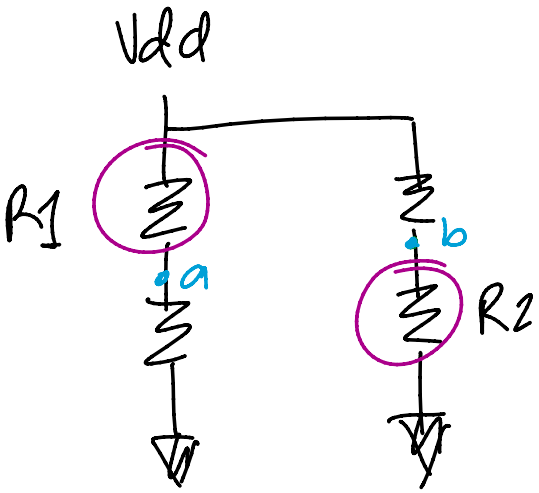
```
#function that sets-up the resistor network as a graph using dictionary
#reference: ChatGPT
def add_edge(adjacency_list, rname, u, v, rvalue):
    if u in adjacency_list:
        if v in adjacency_list[u]:
            adjacency_list[u][v].append({rname: rvalue})
        else:
            adjacency_list[u][v] = [{rname: rvalue}]
    else:
        adjacency_list[u] = {
            v: [{rname: rvalue}]
        }

    if v in adjacency_list:
        if u in adjacency_list[v]:
            adjacency_list[v][u].append({rname: rvalue})
        else:
            adjacency_list[v][u] = [{rname: rvalue}]
    else:
        adjacency_list[v] = {
            u: [{rname: rvalue}]
    }
```



If all conditions are bypassed (series = 1), the two resistors are in series. Else (series = 0), then the two resistors are neither in series nor in parallel.

```
if series == 1:
    print("SERIES")
else:
    print("NEITHER")
```

The logic as to why we use the V2 of resistor 1 is best explained with an example.



Scenario 1: If starting at V2 of R1: path = [a, Vdd, b]
Scenario 2: If starting at V1 of R1: path = [Vdd, b]

Scenario 1 will be detected as a "NEITHER" in the code while scenario 2 will be detected as "SERIES".

● What are the data structures used in your solution? Were these among the common ones we discussed in class or maybe a custom data structure based on one or a combination of these?

The data structures used were the ones discussed in class such as dictionary and lists. The graph was represented using an adjacency list which was implemented using a dictionary. Deque was also used in the shortest path algorithm.

● What are the time and space complexities of your solutions? Do you think these could be improved?

Most of the functions only take O(1) or O(n) time. As for the main loop, at worst case, the resistor network consists of resistors all connected in series and you have to check the first resistor with the last, so the path taken is the longest path possible. In this case, the time complexity will be O(Q*(V+E)) where Q is the number of queries, V is the number of vertices, and E is the number of edges.

The resistor dictionary has two levels of nesting. There are N keys in the outer dictionary and 3 key-value pairs in the inner dictionary. Thus it has a space complexity of O(3N). The adjacency lists has a space complexity of O(V+E). Thus the total space complexity of the program is O(3N + V + E).

I don’t think these complexities can be improved.

MS 2

Saturday, 1 July 2023 1:25 pm

The program for MS 2 builds on the previous MS 1. Additional codes have been added to execute the required functionality for this milestone. In essence, it basically "queries" all possible resistor pair in the network to check for a series or parallel connection between them.

First, we initialize new variables that will be needed.

```
#initializing variables
R = []                  #stores the name of all resistors in the network
Rmarker = []            #also stores the name. purpose is to mark resistors that have connection
parallel_total = 0      #stores the equivalent parallel resistance
series_total = 0        #stores the equivalent series resistance
res_parallel = []       #stores resistors in parallel
                        #will manifest as a nested list if there are many sets of parallel resistors
res_series = []         #stores resistors in series
                        #will manifest as a nested list if there are many sets of series resistors

#loop that stores all the resistor names in R and Rmarker
for resistors in resList: #iterates through all the keys in the resList dictionary where the keys are the resistor names
    R.append(resistors)
    Rmarker.append(resistors)
```

The goal of the block of code on the right is to identify all the pairs of resistors in series or in parallel with each other in the entire network.

```
for i in range(len(R)):
    res1 = R[i]
    if res1 not in Rmarker:
        continue
    else:
        for j in range(i+1, len(R)):
            res2 = R[j]
            if res2 not in Rmarker:
                continue
            else:
                if check_parallel(res1, res2):
                    present = 0
                    for i in range(len(res_parallel)):
                        if res1 in res_parallel[i] and res2 in Rmarker:
                            res_parallel[i].append(res2)
                            Rmarker.remove(res2)
                            present = 1
                            break
                    if present == 0:
                        res_parallel.append([res1,res2])
                        Rmarker.remove(res1)
                        Rmarker.remove(res2)

                elif check_series(res1, res2):
                    present = 0
                    for i in range(len(res_series)):
                        if res1 in res_series[i] and res2 in Rmarker:
                            res_series[i].append(res2)
                            Rmarker.remove(res2)
                            present = 1
                            break
                    if present == 0:
                        res_series.append([res1,res2])
                        Rmarker.remove(res1)
                        Rmarker.remove(res2)
```

Logic:

The resistors will be checked against each other pair by pair using a nested loop. The outer loop goes through all the resistors in the list R starting with the first resistor. The inner loop only goes through all the resistors excluding the ones already checked by the outer loop.
So R1 will be checked against R2, R3,.. Rn
R2 will be checked against R3, R4,... Rn
R3 will be checked against R4, R5,... Rn

First and foremost, it checks if the resistor is in Rmarker. The purpose of this list is to eliminate redundancy and reduce the runtime. There is no need to check every pair of resistors if we already know that one of them is connected in series or parallel to another resistor since no resistor can be both in series and parallel to another resistor.

If it bypasses the initial condition, it then checks for parallel connection first since this test takes less time. It calls the check_parallel function with the name of the two resistors as input. The code of this function is similar to the parallel connection test in the previous milestone.

If the two resistors are in parallel, then we will append the pair as a list into the res_parallel variable and remove the two resistors from Rmarker.

However if the first resistor already belongs to an existing group of parallel connected resistors, then we will append the second resistor to that same group.

All resistors appended to the res_parallel list will be removed from Rmarker so that there will be no need to check for them if we encounter them again in the loop.

The same logic applies to testing for series connection

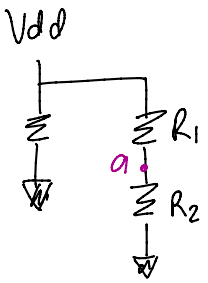
The succeeding block of code extracts the identified resistors and totals their equivalent resistance according to their connection. We should loop through the list as there may be multiple sets of resistors for each connections.

```
for i in range(len(res_series)):
    for k in res_series[i]:
        series_total += resList[k]['resistance']
    if series_total != 0:
        print("[" + ', '.join(res_series[i]) + "]", series_total)
        series_total = 0

for i in range(len(res_parallel)):
    for k in res_parallel[i]:
        parallel_total += 1/resList[k]['resistance']
    if parallel_total != 0:
        print("[" + ', '.join(res_parallel[i]) + "]", int(1/parallel_total))
        parallel_total = 0
```

PROBLEM ENCOUTERED:

This code fails test cases #6, #7,and #8

I suspect the error lies in checking for series connection. The program fails when the resistors are not checked in the "correct" order. In the example circuit below:



R1: {V1 = Vdd, V2 = a}
R2: {V1 = a, V2 = GND}

If we check R1 and R2 where R1 is resistor1 and R2 is resistor 2, taking V1 of R1 and the V2 of R2, we check for a path between "a" and "a". The code evaluates this as a series since the starting vertex is equal to the end vertex.

However if we reverse the order of checking, where R2 is resistor 1 and R1 is resistor 2, the starting vertex is now is V2 of R2 which is "GND" and the ending vertex is V1 of R1 which is "Vdd".

The code evaluates this path as "NEITHER" when it should be "SERIES."

A brute force solution to this problem is to be more mindful of edge cases and add more conditions that checks for this edge case scenario.

Another solution is, instead of only taking one side, to check both vertices of resistor1 for the shortest path to the resistor 2. However this would increase the time complexity of the BFS search by two. To make this more efficient, it is optimal to rectify the graph representation itself. The nodes Vdd and GND do not function similar to the other nodes. They are terminal nodes, meaning that a path which passes on either Vdd or GND is not a series connection. So the implementation of the adjacency list can be made such that Vdd and GND do not have neighbors so the edges should only point away from them.

What are the data structures used in your solution? Were these among the common ones we discussed in class or maybe a custom data structure based on one or a combination of these?

No new data structures were created for this milestone, so it follows the data structures for Milestone 1.

- What are the time and space complexities of your solutions? Do you think these could be improved?

The longest time complexity happens inside the main loop. The worst case scenario would be to check all possible resistor pairs and that all pairs are connected in series with the longest possible path. The time complexity in this case would be O((N^2)*(V+E)).

Measures have been included to safe guard the program from going into its worst case time complexity such as the addition of Rmarker to reduce redundancy in checking resistor pairs.

Since no new data structures are being made inside the main loop, the space complexity remains at O(3N + V + E) where N is the number of resistors.

MS 3

Sunday, 2 July 2023 10:03 am

MS3 builds on the previous milestone MS2. Additional codes have been added to execute the required functionality for this milestone.

It basically operates the same way as the previous milestone except that it loops through the entire network until it is reduced to a single resistor.

The new variable "**mark**" will be discussed further.

```
mark = 0
while len(R) > 1:
    mark += 1
    for i in range(len(R)):
```

Succeeding block is exact code from MS2

INSIDE THE WHILE LOOP:

A new resList variable is initiated to temporarily store the information of the new resistors. It is important not to override the data in the original resList because its values will still be used in the following blocks.

The resistors that remained in Rmarker (resistors with no series or parallel connections) will retain their information and will be added to new_resList.

For the group of resistors in series, their equivalent resistance will be calculated and they will be collapsed into one resistor. The V1 of this new resistor will be the V1 of the first resistor in the list and V2 will be the V2 of the last resistor in the list (due to the order of checking in the loop, the resistors will be naturally arranged in their proper sequence). The variable **mark** serves to eliminate repetition of resistor names across different iterations of the main loop. There is a risk of overriding the information if the new resistor has the same key with an existing resistor in the dictionary even if they are not similar resistors.

The same logic applies to parallel sets.

New_resList will then be transformed into a new resGraph.
Once all the information for the collapsed resistor network is secured in new_resList and new_resGraph, their data will be copied and stored in the original resList and resGraph. The new_resList and new_resGraph will then be cleared of all data. This is for consistency and space optimization, since the variables resList and resGraph are the ones used in the codes above.

The other variables are also cleared of their data in preparation for a new iteration of the loop.

The last block of code in the main loop extracts the resistor names from the resList and appending them to R and Rmarker. The information on R will be used to check for the condition in the main loop.

Recall that the program terminates when the number of resistors is 1 (when lenR == 1). In which case, the resistance of the resistor is printed.

STILL INSIDE WHILE LOOP:

```
    new_resList = {}

    if Rmarker:
        for i in Rmarker:
            r_info = [i, resList[i]['V1'], resList[i]['V2'], resList[i]['resistance']]
            dictResistors(new_resList, r_info)

    for i in range(len(res_series)):
        for k in res_series[i]:
            series_total += resList[k]['resistance']
        if series_total != 0:
            newV1 = resList[res_series[i][0]]['V1']
            newV2 = resList[res_series[i][-1]]['V2']
            newResistance = series_total
            r_info = [f"S{i,mark}", newV1, newV2, newResistance]
            dictResistors(new_resList, r_info)
            series_total = 0

    for i in range(len(res_parallel)):
        for k in res_parallel[i]:
            parallel_total += 1/resList[k]['resistance']
        if parallel_total != 0:
            newV1 = resList[res_parallel[i][0]]['V1']
            newV2 = resList[res_parallel[i][-1]]['V2']
            newResistance = int(1/parallel_total)
            r_info = [f"P{i,mark}", newV1, newV2, newResistance]
            dictResistors(new_resList, r_info)
            parallel_total = 0
```

```
    new_resGraph = {}
    for key in new_resList:
        add_edge(new_resGraph, key, new_resList[key]["V1"], new_resList[key]["V2"], new_resList[key]["resistance"])

    resList = new_resList.copy()
    resGraph = new_resGraph.copy()
    new_resGraph.clear()
    new_resList.clear()

    R = []
    Rmarker = []
    parallel_total = 0
    series_total = 0
    res_parallel = []
    res_series = []

    for resistors in resList:
        R.append(resistors)
        Rmarker.append(resistors)


print(resList[R[0]]['resistance'])
```

PROBLEMS ENCOUNTERED

MS3 failed test case #7 and #8.
The error for this Milestone is similar to the error in the previous milestone: error in checking for series connection. As new resistor values are updated along the way, the order of resistors may become "incorrect" such as in the case of P(0,1) and R3 in the resList example below.

```
{'S(0, 1)': {
    'V1': 'Vdd', 'V2': 'GND', 'resistance': 2000},
 'P(0, 1)': {'V1': 'b', 'V2': 'GND', 'resistance': 500},
 'R3': {'V1': 'Vdd', 'V2': 'b', 'resistance': 1000}}
```

This "disorder" affects the series connection test. In that case, if the program cannot collapse a network to its series connection, since the main loop is a while loop with condition lenR>1, then the number of resistors will never get reduced and thus the loop will continue endlessly.

The same solution provided in MS2 is recommended for MS3.

What are the data structures used in your solution? Were these among the common ones we discussed in class or maybe a custom data structure based on one or a combination of these?

No new data structures were created for this milestone, so it follows the data structures for Milestone 1.

- What are the time and space complexities of your solutions? Do you think these could be improved?

As before, despite the addition of multiple blocks of codes, the worst case scenario remains the same in this milestone. And since we will be enclosing everything in another loop (since we are continuously collapsing the resistor network until it becomes a single resistor) then the time complexity becomes O((N^3)*(V+E)).

Although in essence the data structures inside the main loop are being "updated" not "added" (they do not stack up since they are deleted per iteration), there is an instance where there are two copies of each data structure, so the space complexity becomes O(2*(3N + V + E)) .

This could be improved by taking a new approach to this milestone that would not depend on repetitively collapsing the network to get the equivalent resistance. Unfortunately I cannot come up with a way to do this for now.

MS 4

Sunday, 2 July 2023 10:42 am

For each milestone you have not attempted, answer the following:

● Can you think of a possible solution for this milestone? It could help if you could reference your solutions to previous milestones and enumerate the changes you would need to do.

For my solution, I would build on top of the previous code in MS3. From there, an additional condition, aside from checking for series and parallel, will be added to check for delta connection in the network. The function delta_present should take in only res1 and return a bool value.

A delta connection can be detected as a cycle of 3 vertices such that no two vertices are Vdd and GND.

If there is a series connection inside the delta, this will not be read as a delta connection due to the 3 vertices restriction.

If there is a parallel connection on one branch of the delta, then the parallel connection should first be collapsed. In this case, set the code to continue to the so that the delta to wye transformation happens in the next iteration of the main loop.

Once a delta connection is detected:

Create a new vertex with a name that is not a duplicate and cannot be replicated.

transform the delta connected resistors into wye connected resistors using the formula.

Temporarily store the information on these new resistors in a separate list.

Once the information on these new resistors are complete (name, V1, V2, and resistance), add them to resList, resGraph, and Rmarker and remove the old resistors from the them.

```
            else:
                if check_parallel(res1, res2):
                    present = 0
                    for i in range(len(res_parallel)):
                        if res1 in res_parallel[i] and res2 in Rmarker:
                            res_parallel[i].append(res2)
                            Rmarker.remove(res2)
                            present = 1
                            break
                    if present == 0:
                        res_parallel.append([res1,res2])
                        Rmarker.remove(res1)
                        Rmarker.remove(res2)

                elif check_series(res1, res2):
                    present = 0
                    for i in range(len(res_series)):
                        if res1 in res_series[i] and res2 in Rmarker:
                            res_series[i].append(res2)
                            Rmarker.remove(res2)
                            present = 1
                            break
                    if present == 0:
                        res_series.append([res1,res2])
                        Rmarker.remove(res1)
                        Rmarker.remove(res2)

                #CHECK FOR DELTA CONNECTION HERE

new_resList = {}

if Rmarker:
    for i in Rmarker:
        r_info = [i, resList[i]['V1'], resList[i]['V2'], resList[i]['resistance']]
        dictResistors(new_resList, r_info)
```

● What do you think would be the biggest challenge or step to implement in your proposed solution?

- Detecting cycles may encounter similar problems to the series connection test where some edge cases may not have been considered that would affect the output of the function
- Properly handling the update of information in resGraph and resList considering the data on the new resistors
- Identifying which vertex to retain and which vertex to update in the delta to wye transformation
- Updating the vertices of affected resistors that are not part of the delta connection

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

Considering my pace, I would need an entire day to draft and troubleshoot the code.

● What are the data structures used in your solution? Were these among the common ones we discussed in class or maybe a custom data structure based on one or a combination of these?

No new data structures was created for this milestone, so it follows the data structures for Milestone 1.

● What are the time and space complexities of your solutions? Do you think these could be improved?

A new function that searches for cycles is added in this program. However, it will be designed to terminate after 3 vertices. So evaluating for series connection still has the higher complexity.
Thus, the worst case scenario remains the same in this milestone as the previous which is O((N^3)*(V+E)).

New resList and resGraph are created when transforming from delta to wye, however this will contain at maximum only 3 elements and will also be renewed per iteration of the loop. Thus, the space complexity remains as before with an added constant O(2*(3N + V + E)) + O(1).

This could be improved by taking a new approach to this milestone that would not depend on repetitively collapsing the network to get the equivalent resistance. Unfortunately I cannot come up with a way to do this for now.