Assignment 3

Assign-3 G-2

Sowmya Adunuthula, Nathan Bailey, Yaswadeep Lanka, Akhilandaswari Tulasi Marisetti, Suryansh Patel, Blaine Steck, Kylee Willis

Table of Contents

Introduction

All code is written in Python. It can be found in the Appendix, alongside links to run and examine the code further on Colab and GitHub, respectively. This document will highlight the portions of the code that answer each specific question.

Problem 1: Homogenous Amalgamated Star $S_{n,3}$

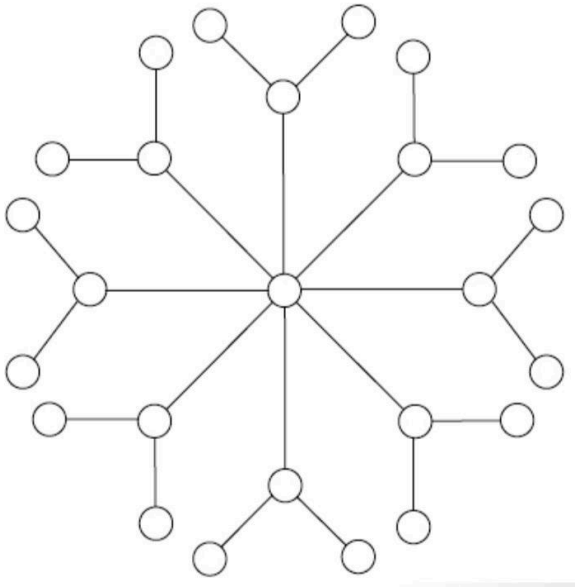

Fig. 1 - A homogeneous amalgamated star graph, its vertices unlabeled.

1. Data Structure

The data structures for this graph are nested arrays. The labels take the form of a list that begins with the centermost vertex and is followed by lists of each individual star/branch in the graph. For example, if *a* is the centermost vertex, *b* is a centroid vertex, and *ba* and *bb* are pendant vertices, the array containing their labels would look like this: *[a, [b, ba, bb]]*.

Each of the weights is found in the same order as the labels, with regards to their source vertex. For example, if *a* is the centermost vertex, *b* is a centroid vertex, and *ba* and *bb* are pendant vertices, the array containing their weights would look like this: *[[a+b, b+ba, b+bb]]*.

There are, in total, 2 sets of nested arrays.

2. Algorithm

```
generate_star(i, n):
      star = []
      if n % 4 == 1:
            if i < ceiling(n/4):
                  append 3*i - 2
```

```
        else:
            append 2*(ceiling(n/4) + i - 1

        for j in range(1,3):
            if i < ceiling(n/4) - 1:
                append j+2
            else if i == ceiling(n/4) and j == 1:
                append 2
            else if i == ceiling(n/4) and j == 2:
                append n - ceiling(n/4) + 3
            else:
                append n + i + j - 2*ceiling(n/4)
    else:
        if i < ceiling(n/4):
            append 3*i - 2
        else:
            append 2*(ceiling(n/4) + i

        for j in range(1,3):
            if i < ceiling(n/4):
                append j+1
            else:
                append n + i + j - 1 - 2*ceiling(n/4)
```

The algorithm we designed for this problem uses the same math as the *Edge irregular k-labeling of amalgamated stars* section from the provided research “Edge irregularity k-labeling for several classes of trees”. As it is a brute force algorithm, each value is calculated and labeled exactly.

In essence, it checks the value of *n* to determine what labeling system needs to be used, then calculates the label of the centroid vertex of one branch/star. The pendant values for that branch are then found. Some pendant values are hard-coded; some are calculated. The centroid and pendant calculations/labels are repeated until all are complete.

The weights are then calculated by looping through each value in the labels array. Each value would be added to either the centermost vertex (if it’s a centroid vertex value) or to its local centroid vertex (if it’s a pendant vertex value). Each calculated value is then appended onto a list of edge weights.

3. Design Strategy

This algorithm is, again, based on the *Edge irregular k-labeling of amalgamated stars*. It therefore follows the brute force design strategy. It can only be applied to a graph where *m=3*.

4. Traversing

Traversing follows a greedy design strategy here, specifically in the form of DFS – each individual star/branch is explored fully before moving on to the next star/branch. The labels are stored the same way – using DFS.

5. Label the Graph

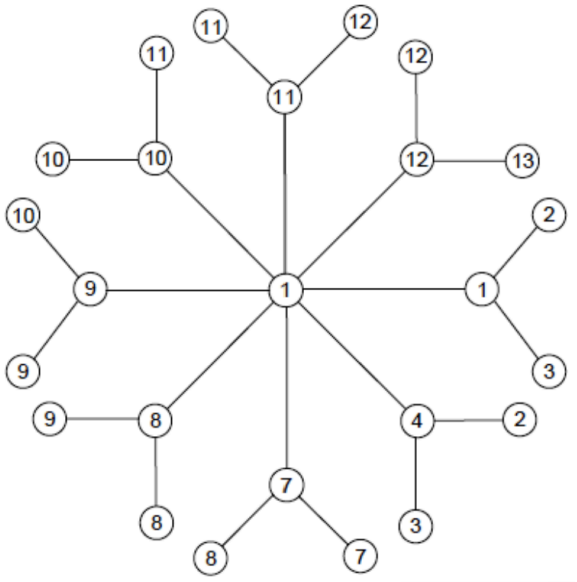

Fig. 2 - A homogeneous amalgamated star graph, its vertices labeled.

Following our algorithm for this problem arrives at the same solution as the provided graph. Below are the results directly from our program output. While it is in a different form (non-graphical output), the output shown in Figure 3 does map out to the same solution as shown above in Figure 2.

```
k value:          13
labels:           [1, [1, 2, 3], [4, 2, 3], [7, 7, 8], [8, 8, 9], [9, 9, 10], [10, 10, 11], [11, 11, 12], [12, 12, 13]]
weights:          [[2, 3, 4], [5, 6, 7], [8, 14, 15], [9, 16, 17], [10, 18, 19], [11, 20, 21], [12, 22, 23], [13, 24, 25]]
is valid labels: True
```

Fig. 3 - The output of running Problem 1.

6. Compare and Tabulate Results

The tabulated results of Problem 1 can be found above in Figure 3. This used the mathematical principles given in the assignment reference material directly. The maximum *k* value, listed here as 13, is calculated directly using the given equation $k = \lceil \frac{3*n+1}{2} \rceil$.

Within the code is a validation function (is_valid), which runs through each vertex to determine that it is no larger than *k*. This algorithm checks the uniqueness of its edge weights as it goes, as well.

7. Maximum Hardware Resources

This algorithm uses predetermined formulas for calculating each vertex, no multithreading or recursive techniques are used. This algorithm will produce accurate results for any given *n* on any given hardware.

8. Time Complexity

The time complexity of this algorithm is O(n). This is from the nested for loops in the weight calculation (calculate_wts), which give the algorithm a time complexity of O(n*m). Since this problem has a definite *m* value (*m=3*), the overall complexity can be simplified to *n*.

Problem 2: Homogeneous Amalgamated Star $S_{n,m}$

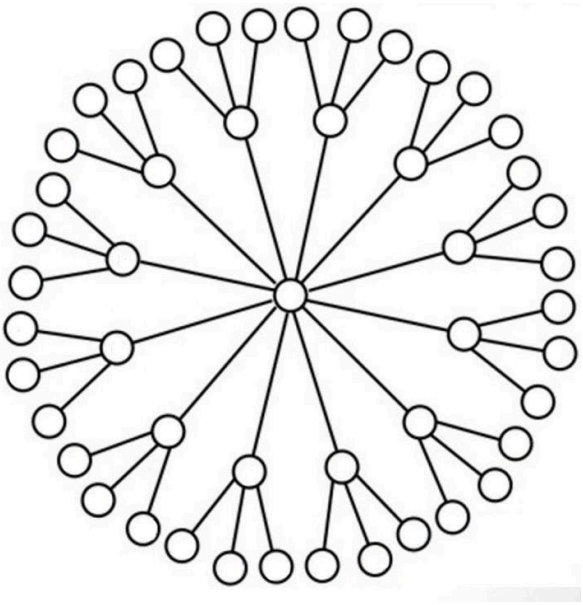

Fig. 4 - A homogeneous amalgamated star graph, its vertices unlabeled.

1. Data Structure

The data structures for this graph are nested arrays. The labels take the form of a list that begins with the centermost vertex and is followed by lists of each individual star/branch in the graph. For example, if *a* is the centermost vertex, *b* is a centroid vertex, and *ba* and *bb* are pendant vertices, the array containing their labels would look like this: *[a, [b, ba, bb]]*.

Each of the weights is found in the same order as the labels, with regards to their source vertex. This is the same type of output as seen in Problem 1. For example, if *a* is the centermost vertex, *b* is a centroid vertex, and *ba* and *bb* are pendant vertices, the array containing their weights would look like this: *[[a+b, b+ba, b+bb]]*.

There are, in total, 2 sets of nested arrays.

2. Algorithm

```
generate_graph(n, m):
      labels = [1, [1]]
      wts = [2]
      v = m*n + 1
      distance = ceiling(v/2) / (n-1)

      for i in range(1, n):
            append floor(d * i) to labels
            append floor(d * i) + 1 to wts

      for i in range(n):
            curr_label = wts[i]
            for j in range(m-1):
                  increase curr_label until its value is not in wts
                  append labels[i+1][0] + curr_label to wts
                  append curr_label to labels[i+1]
```

This algorithm starts with two separate lists: one for the currently used weights (*wts*) and one for the labels of each vertex. Two values are hard-coded into the labels list, *1* - *1*, since the minimum edge weight (*2*) must be met. Their combined value is also hard-coded into the weight array.

From there, the distance between each centroid vertex is calculated (*dist*), and the centroid labels are added to the labels list. The edge weights connected to the centermost vertex are found and added to the weight array.

Each pendant vertex value is then found: this is done by checking each individual edge value against the weights list to see if it has already been used. If it hasn't, it's added to the weight list and the vertex value is added to the labels list.

Just like Problem 1, the weights are then calculated by looping through each value in the labels array. Each value would be added to either the centermost vertex (if it's a centroid vertex value) or to its local centroid vertex (if it's a pendant vertex value). Each calculated value is then appended onto a list of edge weights.

3. Design Strategy

This algorithm, based on *Algorithm 4* from the provided research *Computing Edge Irregularity Strength of Complete M-ary Trees Using Algorithmic Approach*, follows a greedy and dynamic programming design strategy. The value of each centroid vertex is calculated greedily, and the overall structure uses dynamic programming to make sure no edge weights are reused.

4. Traversing

Much like Problem 1, traversing follows a greedy design strategy here, specifically in the form of DFS. Each individual star/branch is explored fully before moving on to the next star/branch. The labels are stored using DFS in the same way.

5. Label the Graph

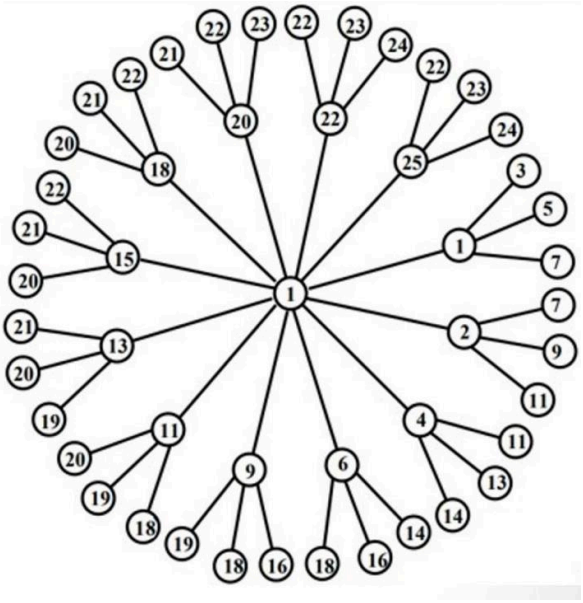

Fig. 5 - A homogeneous amalgamated star graph, its vertices labeled.

Following our algorithm for this problem arrives at the same solution as the provided graph. Below are the results directly from our program output. While it is in a different form (non-graphical output), the output shown in Figure 6 does map out to the same solution as shown above in Figure 5.

```
k value:         25
labels:          [1, [1, 3, 5, 7], [2, 7, 9, 11], [4, 11, 13, 14], [6, 14, 16, 18], [9, 16, 18, 19], [11, 18, 19, 20], [13, 19, 20, 21], [15, 20, 21, 22], [18, 20, 21, 22
], [20, 21, 22, 23], [22, 22, 23, 24], [25, 22, 23, 24]]
weights:         [[2, 4, 6, 8], [3, 9, 11, 13], [5, 15, 17, 18], [7, 20, 22, 24], [10, 25, 27, 28], [12, 29, 30, 31], [14, 32, 33, 34], [16, 35, 36, 37], [19, 38, 39, 40]
, [21, 41, 42, 43], [23, 44, 45, 46], [26, 47, 48, 49]]
is valid labels: True
```

Fig. 6 - The output of running Problem 2.

6. Compare and Tabulate Results

The tabulated results of Problem 2 can be found above in Figure 6. This used the mathematical principles given in the assignment reference material directly. The maximum *k* value, listed here as 25, is calculated directly using the given equation $k = \lceil \frac{m*n+1}{2} \rceil$.

Within the code is a validation function (is_valid), which runs through each vertex to determine that it is no larger than *k*. This algorithm checks the uniqueness of its edge weights as it goes, as well.

7. Maximum Hardware Resources

As no multithreading or recursive techniques are used in calculating the values of vertices, this algorithm will produce accurate results at any value *n* and *m* given enough resources are available.

8. Time Complexity

The time complexity of this algorithm is O(n*m), again from the nested for loops.

Problem 3: N branches with Centroid Vertex

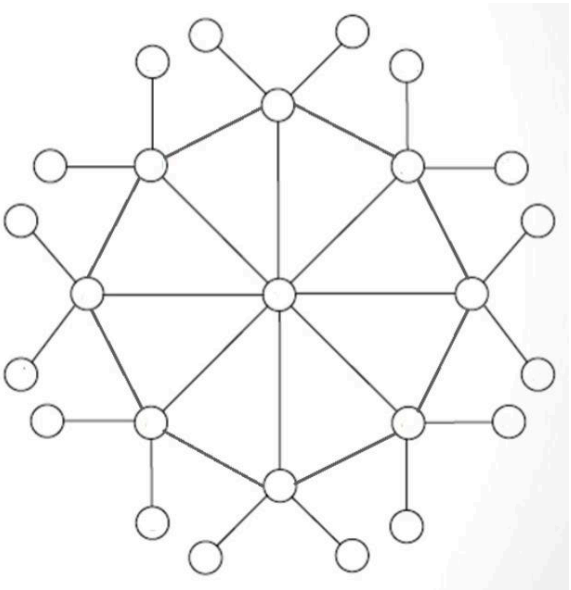

Fig. 7 - A homogeneous trampoline graph, its vertices unlabeled.

1. Name

Our chosen name for this star graph is the “Homogeneous Trampoline”. The centroid vertices look like the trampoline itself, while the pendant vertices resemble the springs.

2. Formulae for Order and Size

The order of a homogeneous trampoline graph is determined using the formula $|V| = m * n + 1$, the same formula used in Problem 2’s amalgamated star graph. The size of a trampoline graph is found using the formula $|E| = (m + 1) * n$.

The difference in the size formula is due to the trampoline graph having extra edges between each centroid vertex – that is, there is one extra edge for each centroid vertex, meaning that the trampoline sees *n* extra edges as compared to the homogeneous amalgamated star graph.

3. Data Structure

The data structures for this graph are nested arrays. The labels take the form of a list that begins with the centermost vertex and is followed by lists of each individual star/branch in the graph. For example, if *a* is the centermost vertex, *b* is a centroid vertex, and *ba* and *bb* are pendant vertices, the array containing their labels would look like this: *[a, [b, ba, bb]]*.

Each of the non-cyclical weights is found in the same order as the labels, with regards to their source vertex. This is the same type of output as seen in Problems 1 and 2. For example, if *a* is the centermost vertex, *b* is a centroid vertex, and *ba* and *bb* are pendant vertices, the array containing their weights would look like this: *[[a+b, b+ba, b+bb]]*.

All cyclical weights are calculated afterwards, and they are stored with the vertices that they connect to inside of a tuple. For example, if *b* and *c* are centroid vertices that are next to each other, the array containing their cyclical weights would look like this: *[a+b, (a, b)]*.

There are, in total, 3 arrays, two of which have nested arrays and one of which has a tuple nested inside an array.

4. Assign the Labels

The maximum label value, *k*, for the trampoline graph can be calculated using this formula: $k = \lceil \frac{(m+1)*n+1}{2} \rceil$, or $k = \lceil \frac{|E|+1}{2} \rceil$. For the example problem given in the slides, that means that the maximum *k* value is 17.

The center of the graph should be labeled first: it should be labeled with that *k* value. The half (ceiling; inclusive) centroid vertices of the graph should then be labeled, beginning with a value of *1* and increasing until *k-1* as necessary. The remaining half (ceiling; exclusive) of the centroid vertices should be labeled afterwards, beginning with the value *k-1* and decreasing to *1* as necessary. All edge weights calculated with those labels should be stored and checked against to make sure no repeats occur. If a repeat edge weight occurs, that label should be skipped and incremented/decremented, depending on the half of the circle.

The output of this algorithm can be seen in the figure below in task 5.

5. Label the Graph

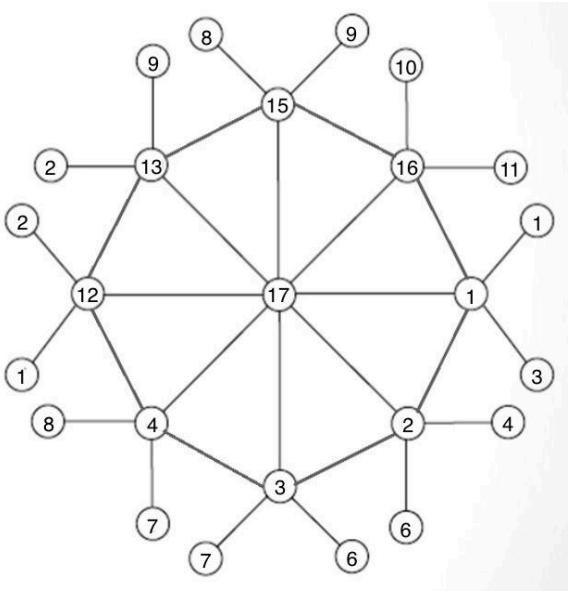

Fig. 8 - A homogeneous trampoline graph, its vertices labeled.

When executed, this trampoline graph would be labeled as seen above in Figure 8. Of course, it can be applied to more graphs with various *n* and *m* values, but for the purposes of this assignment, the above figure was filled in.

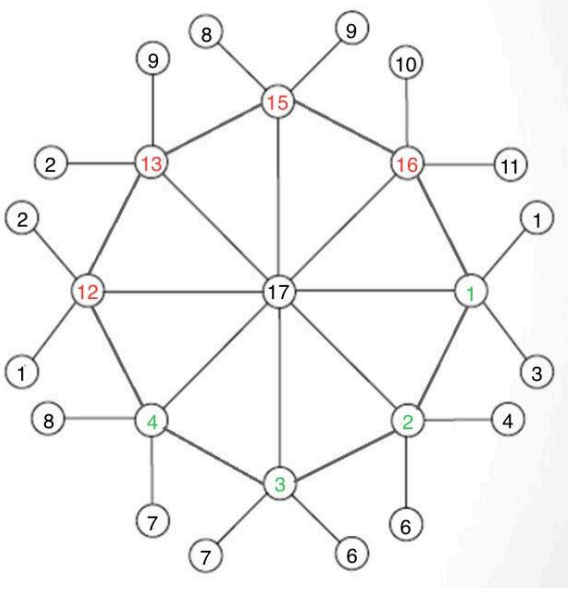

Fig. 9 - A homogeneous trampoline graph, its vertices labeled and in color.

Figure 9 shows more clearly the split between the two centroid loops. The green numbers are found in the first loop, which increases from *1* to *k-1*. The red numbers are found in the second loop, which decreases from *k-1* to *1*.

The output of running this algorithm is shown below in Figure 10.

```
k value:         17
labels:          [17, [1, 1, 3], [2, 4, 6], [3, 6, 7], [4, 7, 8], [12, 1, 2], [13, 2, 9], [15, 8, 9], [16, 10, 11]]
weights:         [[18, 2, 4], [19, 6, 8], [20, 9, 10], [21, 11, 12], [29, 13, 14], [30, 15, 22], [32, 23, 24], [33, 26, 27]]
cycle weights:   [[3, (1, 2)], [5, (2, 3)], [7, (3, 4)], [16, (4, 12)], [25, (12, 13)], [28, (13, 15)], [31, (15, 16)], [17, (16, 1)]]
is valid labels: True
```

Fig. 10 - The output of running Problem 3.

6. Pseudocode

```
generate_trampoline(n, m, k, e):
      labels = [k]
      weights = [e+1 values of false]
      mid = ceiling(n/2)
      for i in range 1 to mid:
            for j in range i to k:
                  if it's not used:
                        add j to labels
                        set weights[j+k] = true
                        Break
      for i in range n to mid:
            for j in range k-1 to i-1:
                  if it's not used:
                        add j to labels
                        set weights[j+k] = True
                        break

      prev = 2
      for i in range 0 to n:
            for j in range 0 to m-1:
                  avail = -1
                  for z in range prev to e:
                        if z is unused:
                              avail = z
                              prev = avail +1
                  calc = avail - centroid label
                  weights[avail] = True
                  append calc to labels[i+1]
```

See "Assign the Labels" for a detailed explanation.

Appendix

All code contained in this appendix can be run on Colab. It can also be found on GitHub in branches “assignment/suryansh” and “problem3”. Each of the code sections below are provided for validation/proof, if necessary.

Problem 1 Code - problem_1.py

```
import math

def generate_star(i , n):
    star = []
    if n % 4 == 1:
        if 1 <= i <= math.ceil(n/4):
            star.append(3*i - 2)
        elif math.ceil(n/4) + 1 <= i <= n:
            star.append(2*(math.ceil(n/4)) + i - 1)
        else:
            print('for center of star case not satisfy - 1')

        for i in range(1,3):
            if 1 <= i <= math.ceil(n/4) - 1:
                star.append(j+2)
            elif i == math.ceil(n/4) and j == 1:
                star.append(2)
            elif i == math.ceil(n/4) and j == 2:
                star.append(n - (math.ceil(n/4)) + 3)
            elif math.ceil(n/4) + 1 <= i <= n:
                star.append(n+i+j-2*(math.ceil(n/4)))
            else:
                print('invalid pendent parameters')
    else:
        if 1 <= i <= math.ceil(n/4) + 1:
            star.append(3*i - 2)
        elif math.ceil(n/4) + 2 <= i <= n:
            star.append(2*(math.ceil(n/4)) + i)
        else:
            print('for center of star case not satisfy - 2')

        for j in range(1,3):
            if 1 <= i <= math.ceil(n/4):
                star.append(j+1)
            elif math.ceil(n/4) + 1 <= i <= n:
                star.append(n+i+j-1-2*(math.ceil(n/4)))
```

```
            else:
                print('invalid pendent label')

    return star

def generate_graph(n):
    lables = [1]
    if n < 3:
        print('invalid args n = ',n)
        return lables

    for i in range(1,n+1):
        lables.append(generate_star(i,n))

    return lables

def calculate_wts(arr):
    wt = []
    for lab in arr[1:]:
        temp = [1+lab[0]]
        for num in lab[1:]:
            temp.append(lab[0] + num)
        wt.append(temp)
    return wt

def flatten(arr):
    flattened = []
    for item in arr:
        if isinstance(item, list):
            flattened.extend(flatten(item))
        else:
            flattened.append(item)
    return flattened

def is_valid(arr, k):
    flattened = flatten(arr)
    return k >= max(flattened)


# main
if __name__ == "__main__":
    n = 8
    vertex = generate_graph(n)
    wts = calculate_wts(vertex)
```

```
k = math.ceil((3*n + 1)/2)
print('k value',k)
print('lables' , vertex)
print('weights :', wts )
print('is valid lables:',is_valid(vertex,k))
```

Problem 2 Code - problem_2.py

```
from problem_1 import calculate_wts, is_valid
import math

def generate_graph(n,m):
    lables = [1, [1]]
    wts = [2]
    v = m*n + 1
    d = math.ceil(v/2) / (n - 1) # 2.272727 in our case

    # generate edges connected to center
    for i in range(1,n):
        value = math.floor(d*i)
        lables.append([value])
        wts.append(value + 1)

    # generate pendent lables
    for i in range(n):
        assumed_label = min(wts) # wts[i]

        for j in range(m-1):
            while (lables[i+1][0] + assumed_label) in wts:
                assumed_label += 1
            wts.append(lables[i+1][0] + assumed_label)
            lables[i+1].append(assumed_label)

    return lables , wts

# main
if __name__ == "__main__":
    m = 4
    n = 12
    k = math.ceil((m*n + 1)/2)
    labels,wts = generate_graph(n,m)
    print('k value',k)
    print('labels :',labels)
    print('weights :',calculate_wts(labels))
    print('is valid lables:',is_valid(labels,k))
```

Problem 3 Code - problem_3.py

```
from problem_1 import is_valid
import math

def generate_trampoline(n,m,k,e):
    labels = [k]
    used_weights = []


    used_weights = [False] * (e + 1)

    # generate labels and edges connected to center
    # first half of graph; work forwards from 0 to midpoint (inclusive)
    mid = math.ceil(n/2)
    prev = -1
    for i in range(1,mid+1):
        for j in range(i, k):
            if not used_weights[j + k] and (i == 1 or (i != 1 and not used_weights[prev + j])):
                labels.append([j])
                used_weights[j + k] = True
                if (prev != -1):
                    used_weights[j + prev] = True
                prev = j
                break

    # second half of graph; work backwards from n to midpoint (exclusive)
    prev = labels[1][0]
    for i in range(n, mid, -1):
        for j in range(k-1, i-1, -1):
            if not used_weights[j + k] and not used_weights[prev + j]:
                if i != mid+1 or (i == mid+1 and not used_weights[labels[mid][0] + j]):
                    labels.insert(mid+1,[j])
                    used_weights[j + k] = True
                    used_weights[j + prev] = True
                    if i == mid+1:
                        used_weights[labels[mid][0] + j] = True
                    prev = j
                    break

    # generate pendent labels
    prev = 2
    for i in range(n):
        for j in range(m-1):
            #run through all available edges; pull the lowest out and use it on the next pendant
```

```
            low_avail = -1
            for z in range(prev,e):
                if not used_weights[z]:
                    low_avail = z
                    prev = low_avail + 1
                    break
            if low_avail == -1: # error; not all weights could be used
                return labels, used_weights

            #find pendant vertex value
            calc = low_avail - labels[i+1][0]
            used_weights[low_avail] = True
            labels[i+1].append(calc)

    return labels , used_weights

def calculate_wts(arr):
    #normal weights -- those found in this problem, without edges between the centroid vertices
    wt = []
    for label in arr[1:]:
        temp = [arr[0]+label[0]]
        for num in label[1:]:
            temp.append(label[0] + num)
        wt.append(temp)

    #centroid edge weights
    cent = []
    prev = arr[1][0]
    for label in arr[2:]:
        temp = prev + label[0]
        cent.append([temp , (prev , label[0])])
        prev = label[0]
    cent.append(arr[-1][0] + arr[1][0], (arr[-1][0], arr[1][0]))
    return wt, cent


# main
if __name__ == "__main__":
    m = 3
    n = 8
    k = math.ceil(((m+1)*n + 1)/2)
    e = (m+1)*n + 1 # to find total no of edges including centroid
    # v = m*n + 1 # total no of vertex in graph
    labels,wts = generate_trampoline(n,m,k,e)
```

```
print('k value:\t',k)
print('labels:\t\t',labels)
standard, centroid_cycle = calculate_wts(labels)
print('weights:\t',standard)
print('cycle weights:\t', centroid_cycle)
print('is valid labels:',is_valid(labels,k))
```